版权注意事项：
1、书籍版权归作者和出版社所有
2、本PDF仅限用于个人获取知识，进行私底下的知识交流
3、PDF获得者不得在互联网上以任何目的进行传播
4、如觉得书籍内容很赞，请购买正版实体书，支持作者
5、请于下载PDF后24小时内删除本PDF。

HZ BOOKS
华章IT

Kotlin 极简教程

陈光剑◎编著

Kotlin 快速入门教程，零基础学会 Kotlin 开发

从 Kotlin 基础知识到动手实战，包含了大量精选示例代码和应用案例

机械工业出版社
China Machine Press

内容简介

本书从 Kotlin 的 HelloWorld 开始，生动讲解 Kotlin 的基础知识和实战案例。全书共 16 章，分为三大部分：第一部分“Kotlin 语言基础”包括第 1~4 章，介绍如何快速开始使用 Kotlin，以及 Kotlin 中的语法、基本数据类型和类型系统；第二部分“编程进阶”包括第 5~10 章，介绍 Kotlin 标准库中的集合类、泛型、面向对象编程特性、函数式编程、协程、与 Java 的互操作；第三部分“综合实战”包括第 11~16 章，介绍如何使用 Kotlin 集成 Spring Boot、Spring MVC 等框架来开发 Web 服务端应用，如何使用 Kotlin 开发 Android 应用，如何使用 Kotlin 的 DSL 代码风格，如何操纵文件 IO、正则表达式以及多线程，如何用 Kotlin Native 与 C 语言互操作。

Kotlin
极简教程

陈光剑◎编著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Kotlin 极简教程 / 陈光剑编著 . —北京：机械工业出版社，2017.9

ISBN 978-7-111-57993-9

I. K… II. 陈… III. JAVA 语言 – 程序设计 – 教材 IV. TP312.8

中国版本图书馆 CIP 数据核字（2017）第 220693 号

Kotlin 极简教程

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：吴 怡　　责任校对：李秋荣

印　　刷：北京市荣盛彩色印刷有限公司　　版　　次：2017 年 9 月第 1 版第 1 次印刷

开　　本：186mm × 240mm 1/16　　印　　张：31.75

书　　号：ISBN 978-7-111-57993-9　　定　　价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

版权所有 • 侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

Preface 序　言

首先非常感谢我的同事陈光剑在图书还没有正式发布之前能将所有章节发给我阅读，并邀请写一个序言介绍这本书。Kotlin 1.0 在 2016 年初推出后，受到非常多的程序员认可，当然你也可以理解为热捧。越来越多的项目开始使用 Kotlin 开发，而且很多技术和框架都开始支持 Kotlin，其中最知名的就是 Google 宣布 Kotlin 为 Android 开发官方语言，其他如 Spring Framework 5.0 原生支持 Kotlin，Spring Boot 2、JUnit 5 等都支持 Kotlin。众多的开发者非常看重 Kotlin 带来的便捷性和代码精简，但是 DSL、协程 (CoRoutine)、Reactive 等高级特性越来越多地被采用。在这里我就不介绍 Kotlin 众多特性了，本书会一一呈现给大家。

学习一门语言一定会涉及图书和资料，本书是不可多得的优秀图书，内容翔实，描述简洁，对编程语言有深刻的理解，融入了自己多年的开发经验。不可否认，Kotlin 官方文档写得非常不错，还有 Safari Online Books 上的 Hadi Hariri 的 Kotlin 视频教程也很好。但是对于学习一项新技术来说，快速掌握、系统了解、理解不要出现偏差，非常重要，如果有这样一本中文图书，对中国程序员会帮助非常大，大家可以花更少的时间了解和学习 Kotlin，并在实际的开发中快速使用起来。

Kotlin 非常新，而且发展也非常快，是一门非常有前景的开发语言，我个人希望 Android 和 Java 程序员都能停下来阅读这本书，系统地了解这项技术。当然没有一本书能够跟得上技术快速发展的步伐，新特性、技术实现总是快速地推陈出新，但是知识原理都是相通的。DSL 设计可能很难，但是你如果非常理解书中讲到的 Function Literals with Receiver、Invoke Operator 和 Infix notation，配合基础的 Lambda、Operator overloading 和 Function Extension，那么一切就非常简单啦。

我这一年也在做一些 Kotlin 相关的研究，主要集中于 DSL 设计、Reactive、CoRoutine 和相关框架整合等，但是真没有能力和精力来写一本关于 Kotlin 的图书帮助到程序员，因为写本书相当不容易，写本书好书更加烧脑费时，令人佩服的是我的同事陈光剑给大家带来了好书，期待这本书给更多的人带来实惠，使更多的人能够快速学习和使用 Kotlin。

雷卷，阿里巴巴 70 后 Java 程序员

2017 年 8 月于杭州

前　言 *Preface*

Kotlin 是 JetBrains 团队开发的一门现代的、注重工程实用性的静态类型编程语言，JetBrains 团队以开发了世界上最好用的 IDE 而著称。Kotlin 于 2010 年推出，并在 2011 年开源。Kotlin 充分借鉴并汲取了 Java、Scala、Groovy、C#、Gosu、JavaScript、Swift 等多门杰出语言的优秀特性，语法简单优雅、表现力丰富、抽象扩展方便、代码可重用性好，同时也支持面向对象和函数式编程的多范式编程。Kotlin 可以编译成 Java 字节码运行在 JVM 平台和 Android 平台，也可以编译成 JavaScript 运行在浏览器环境，而且还可以直接编译成机器码的系统级程序，直接运行在嵌入式、iOS、MacOS/Linux/Windows 等没有 JVM 环境的平台。Kotlin 源自产业界，它解决了工程实践中程序设计所面临的真实痛点，例如，类型系统可以避免空指针异常的问题。

我最早是被 Kotlin 的下面这段代码所吸引：

```
package com.easy.kotlin

fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}

fun isOdd(x: Int) = x % 2 != 0
fun length(s: String) = s.length
fun main(args: Array<String>) {
    val oddLength = compose(::isOdd, ::length)
    val strings = listOf("a", "ab", "abc")
    println(strings.filter(oddLength))
}
```

13 行。

这大约是在三年前，当时我在学习 Java 8 中的函数式编程以及 Lambda 表达式等新特性。那时，我也对 Scala、Groovy、Clojure、Haskell 等技术很感兴趣，在伴随着兴趣的学习过程中，我无意中看到了上面那段 Kotlin 代码，第一眼看到这么优雅的函数式编程风格，尤其是 compose 函数的定义实现，深深地吸引了我。

Swift 使用 func 关键字声明函数多个 c，怪怪的；Groovy、Scala 等语言使用 def 关键字，与函数本义联想不直接；JavaScript 使用 function 关键字又显得死板了些。而 Kotlin 中的 fun

则简单优雅地恰到好处，关键还让人自然联想到“乐趣、开心、愉快”这样的意思，使用Kotlin每写一个函数都是充满乐趣的。

我们不妨来看看同样的逻辑实现，如果使用Java 8来写会如何：

```
package com.easy.kotlin;

import java.util.ArrayList;
import java.util.List;

interface G<A, B> {
    B apply(A a);
}

interface F<B, C> {
    C apply(B b);
}

interface FG<A, B, C> {
    C apply(A a);
}

public class ComposeFunInJava<A, B, C> {
    public static void main(String[] args) {
        G<String, Integer> g = (s) -> s.length();
        F<Integer, Boolean> f = (x) -> x % 2 != 0;
        FG<String, Integer, Boolean> fg = (x) -> f.apply(g.apply(x));

        List<String> strings = new ArrayList();
        strings.add("a");
        strings.add("ab");
        strings.add("abc");
        List<String> result = new ArrayList();
        for (String s : strings) {
            if (fg.apply(s)) {
                result.add(s);
            }
        }
        System.out.println(result);
    }
}
```

36行，差不多是Kotlin的3倍。

我们知道，Java是一门非常优秀的面向对象语言。但是在函数式编程方面，与其他函数语言相比，还是显得有些笨重与生涩，并且其内在体现出来的思想，依旧是面向对象的思想。

功能强大的Scala语言，复杂性相对较高，学习成本也高，而且Scala与Java的互操作性没有Kotlin好。所以，如果既想方便、流畅地使用Java强大与完善的生态库，又想使用更加先进的编程语言特性，无疑Kotlin是个非常不错的选择。

因此，我立马进入了 Kotlin 的世界！

Kotlin 之前一直是默默无闻的，直到今年（2017 年）Google IO 大会上，Google 宣布正式支持 Kotlin 为 Android 的官方开发语言，而且从 Android Studio 3.0 开始，将直接内置集成 Kotlin 而无需安装任何的插件。另外，在 Spring 5.0 M4 中也引入了对 Kotlin 专门的支持。

在学习和使用 Kotlin 的过程中，我发现我越来越喜欢 Kotlin，它是一门非常优秀、优雅、有趣、流畅实用的语言，绝对值得一试。感谢 Kotlin 团队！

本书可以说是我对 Kotlin 的使用和思考过程的粗浅总结。通过本书的写作，我加深了对 Kotlin 语言及其编程的理解，深刻体会到了学无止境的含义。写书的过程也是我系统学习与思考 Kotlin 的过程，如果本书能够对你有所帮助，将不胜欣慰。

如何阅读本书

受到 Kotlin 极简特性的感召，我希望也通过极简的表述，全面介绍 Kotlin 语言特性以及如何使用 Kotlin 进行实际项目开发。全书共 16 章，分为三大部分：第一部分“Kotlin 语言基础”包括第 1~4 章，介绍如何快速开始使用 Kotlin，以及 Kotlin 语法、基本数据类型和类型系统；第二部分“编程进阶”包括第 5~10 章，介绍 Kotlin 标准库中的集合类、泛型、面向对象编程特性、函数式编程、协程、与 Java 的互操作；第三部分“综合实战”包括第 11~16 章，介绍如何使用 Kotlin 集成 Spring Boot、Spring MVC 等框架来开发 Web 服务端应用，如何使用 Kotlin 开发 Android 应用，如何使用 Kotlin 的 DSL 代码风格，如何操纵文件 IO、正则表达式以及多线程，如何定义 Kotlin Native 与 C 语言互操作。

建议初学者最好按照顺序来阅读本书，Java 程序员可以从第 5 章开始阅读，直接研究 Kotlin 的特殊点，C 语言程序员可以先读第 16 章再考虑补充前面的基础知识。

本书章节内容如下。

第一部分 “Kotlin 语言基础”

第 1 章是 Kotlin 语言的简介，带领读者快速学习 Kotlin 的环境搭建以及常用工具的使用。该章最后还给出一个编程语言学习的小结。通过该章的学习，能够快速进入 Kotlin 的世界。

第 2 章是快速开始 Hello World，分别给出了使用命令行 REPL、可执行应用程序、Web RESTful、Android、JavaScript 等平台环境上的 HelloWorld 示例。

第 3 章介绍 Kotlin 语言的基础知识，包括 Kotlin 语言的关键字与标识符等、表达式与流程控制、运算操作符、函数及其扩展等基本内容。

第 4 章介绍 Kotlin 语言的基本类型和类型系统。包括 Kotlin 的内置基本类型：数字、字符串、布尔、数组等，以及 Kotlin 中引入的特殊的可空类型。

第二部分 “编程进阶”

第 5 章介绍 Kotlin 标准库中的集合类：List、Set、Map。通过该章的学习，我们将了解到 Kotlin 是如何扩展 Java 集合库的，这使得写代码更加简单容易。

第 6 章介绍 Kotlin 泛型的基本概念、型变以及类型边界等内容。

第 7 章介绍 Kotlin 面向对象编程的特性：类与构造函数、抽象类与接口、继承以及多重继承等基础知识，同时介绍了 Kotlin 中的注解类、枚举类、数据类、密封类、嵌套类、内部类、匿名内部类等。

第 8 章介绍 Kotlin 函数式编程的相关内容，其中重点介绍了 Kotlin 中的高阶函数、Lambda 表达式、闭包等核心语法，并给出相应的实例说明。还探讨了关于 Lambda 演算、Y 组合子与递归等函数式编程思想的内容。

第 9 章介绍 Kotlin 中的协程，包括协程的创建、执行、取消等操作。在该章的后半部分，主要探讨挂起函数的组合执行、协程上下文与调度器、通道与管道等相关内容。最后，对协程与线程进行了简单比较，简要介绍了 Kotlin 的协程 API 库。

第 10 章介绍 Kotlin 与 Java 的互操作。

第三部分 “综合实战”

第 11 章介绍如何使用 Kotlin 集成 Spring Boot、SpringMVC 等框架来开发 Web 服务端应用，给出了一个完整的开发实例。最后，简单介绍了 Spring 5.0 中对 Kotlin 的支持特性。

第 12 章介绍使用 Kotlin 集成 Gradle 开发的相关内容。

第 13 章介绍如何使用 Kotlin 开发 Android 应用的具体方法，其中用到了 Anko、ButterKnife、Realm 等相关框架。

第 14 章介绍 Kotlin 中 DSL 的相关内容。我们将会看到 Kotlin 的扩展函数和高阶函数（Lambda 表达式）特性，为定义 Kotlin DSL 提供了极大的支持。使用 DSL 的代码风格，可以让程序更加直观易懂、简洁优雅。

第 15 章介绍 Kotlin 文件 IO 操作、正则表达式以及多线程相关的内容。

第 16 章简单介绍了 Kotlin Native，并给出了 Kotlin 与 C 语言互操作的完整实例。

谁适合阅读本书

本书适合于所有程序员，不管你是前端开发者、Android/iOS 开发者，还是 Java 开发者、C 语言开发者，学习 Kotlin 都会非常容易，本书将帮助读者快速进入 Kotlin 世界。

如果你目前还不是程序员，但想进入编程世界，那么可以尝试从 Kotlin 开始学习。虽然本书中的部分内容需要一定的 Java 编程基础，但是 Kotlin 本身的极简特性能激发你对编程的兴趣。

代码下载

几乎在每章末尾处，我都附上了该章示例工程源代码地址。这些源码都在 https://github.com/EasyKotlin。读者可以根据需要，自由下载学习。

致谢

在本书的写作出版过程中，得到了很多人的帮助和陪伴。

首先要感谢的是我的妻子和两个可爱的孩子。正是有了你们的陪伴，我的生活才更加有意义。我要感谢我的父母，虽然他们可能不知道我写的东西是什么，但是我在写书过程中，脑海里会经常浮现出他们的笑脸，正是由于他们的辛勤养育，我才能长成今天的我。

我要衷心地感谢本书的策划编辑吴怡女士。在本书的写作修改过程中，她耐心细致地对稿件进行了详尽审阅和批注，还提出了很多宝贵的修改建议，并不断给予我鼓励，才使我快速完成了这本书。还感谢审稿编辑陈佳媛、曲熠对本书提出的宝贵意见，同时感谢本书出版过程中所有付出辛劳的华章公司工作人员。

在此，我还要特别感谢我们公司的技术大牛雷卷（陈立兵），非常感谢他能够抽出宝贵时间审阅本书，并给出了内容的勘误，极大地提升了本书的质量，最后，还为本书写了序。真的非常感谢！

我还要感谢在我的工作学习中认识的所有朋友和同事们，能够认识你们并跟你们一起学习、共事，是我的荣幸！

请联系我

虽然在本书写作与修改的过程中，我竭尽全力追求简单正确、清晰流畅地表达内容，但是限于自身水平和有限的时间，仍会有错误与疏漏之处，还望各位读者不吝指正。

对本书的任何问题、意见或者建议，欢迎通过邮件 universsky@163.com 与我交流。

快乐生活，快乐学习，快乐分享，快乐实践出真知。祝大家阅读愉快！

陈光剑

2017 年 8 月于杭州

Contents 目　录

序　言

前　言

第一部分　Kotlin 语言基础

第 1 章　Kotlin 简介 ····· 2

1.1　Kotlin 简史 ····· 2

1.1.1　Kotlin 概述 ····· 3

1.1.2　Kotlin 元年：2016 ····· 3

1.2　快速学习工具 ····· 4

1.2.1　云端 IDE ····· 4

1.2.2　本地命令行环境搭建 ····· 5

1.2.3　使用 IntelliJ IDEA ····· 9

1.2.4　使用 Eclipse ····· 13

1.2.5　使用 Gradle 构建 Kotlin 项目 ··· 17

1.2.6　编程语言学习小结 ····· 18

1.3　本章小结 ····· 20

第 2 章　快速开始 HelloWorld ····· 21

2.1　命令行版 HelloWorld ····· 21

2.2　应用程序版 HelloWorld ····· 21

2.3　Web RESTful HelloWorld ····· 24

2.4　Android 版的 HelloWorld ····· 34

2.5　JavaScript 版 HelloWorld ····· 39

2.6　本章小结 ····· 49

第 3 章　Kotlin 语言基础 ····· 50

3.1　包 ····· 50

3.2　声明变量和值 ····· 56

3.3　变量类型推断 ····· 57

3.3.1　省去变量类型 ····· 57

3.3.2　使用 is 运算符进行类型检测 ···· 59

3.4　字符串与其模板表达式 ····· 59

3.5　流程控制语句 ····· 60

3.5.1　if 表达式 ····· 60

3.5.2　when 表达式 ····· 62

3.5.3　for 循环 ····· 64

3.5.4　while 循环 ····· 64

3.5.5　break 和 continue ····· 65

3.5.6　return ····· 66

3.5.7　标签 ····· 69

3.5.8　throw 表达式 ····· 71

3.6　代码注释 ····· 72

3.7　语法与标识符 ····· 73

3.7.1　修饰符 ····· 73

3.7.2　关键字（保留字） ····· 75

3.7.3 操作符和操作符的重载 …… 81
3.7.4 操作符优先级 …… 82
3.7.5 一元操作符 …… 83
3.7.6 二元操作符 …… 85
3.8 扩展函数和扩展属性 …… 90
3.8.1 扩展函数 …… 90
3.8.2 扩展属性 …… 93
3.9 空指针安全 …… 93
3.10 标准库 API 简介 …… 96
3.11 本章小结 …… 97

第 4 章 基本数据类型与类型系统 …… 98

4.1 什么是类型 …… 98
4.2 编译时类型与运行时类型 …… 100
4.2.1 弱类型与强类型 …… 100
4.2.2 静态类型与动态类型 …… 101
4.2.3 显式类型与隐式类型 …… 101
4.3 根类型 Any …… 101
4.4 基本类型 …… 102
4.4.1 数字类型 …… 104
4.4.2 字符类型与转义符 …… 109
4.4.3 布尔类型 …… 111
4.4.4 字符串类型 …… 112
4.4.5 数组类型 …… 117
4.5 可空类型 …… 120
4.5.1 Kotlin 中的 null …… 120
4.5.2 可空类型 String? 与安全调用 ?. …… 122
4.5.3 可空性的实现原理 …… 124
4.5.4 可空类型层次体系 …… 126
4.6 kotlin.Unit 类型 …… 127
4.7 kotlin.Nothing 类型 …… 128
4.8 类型检测与类型转换 …… 131
4.8.1 is 与 !is 运算符 …… 131
4.8.2 as 运算符 …… 133
4.9 本章小结 …… 134

第二部分 编程进阶

第 5 章 集合类 …… 136

5.1 集合类的基本概念 …… 136
5.1.1 集合类是一种数据结构 …… 137
5.1.2 集合类是 ADT …… 137
5.1.3 连续存储和离散存储 …… 138
5.2 Kotlin 集合类简介 …… 138
5.3 List …… 139
5.3.1 创建不可变 List …… 140
5.3.2 创建可变集合 MutableList …… 142
5.3.3 遍历 List 元素 …… 143
5.3.4 List 元素操作函数 …… 146
5.3.5 List 集合类的基本运算函数 …… 151
5.3.6 过滤操作函数 …… 159
5.3.7 映射操作函数 …… 166
5.3.8 分组操作函数 …… 169
5.3.9 排序操作符 …… 170
5.3.10 生产操作符 …… 171
5.4 Set …… 174
5.4.1 空集 …… 174
5.4.2 创建 Set …… 176
5.4.3 使用 Java 中的 Set 类 …… 177
5.4.4 Set 元素的加减操作 plus minus …… 179
5.5 Map …… 179
5.5.1 创建 Map …… 180
5.5.2 访问 Map 的元素 …… 184

5.5.3 Map 操作符函数 …… 186
5.6 本章小结 …… 192
第 6 章 泛型 …… 193
6.1 泛型简介 …… 193
6.1.1 泛型与多态 …… 193
6.1.2 为何引进泛型 …… 194
6.1.3 Java 与 Kotlin 中的泛型 …… 195
6.2 型变 …… 197
6.2.1 Java 的类型通配符 …… 197
6.2.2 泛型和数组的型变 …… 201
6.3 Kotlin 中的泛型 …… 205
6.3.1 out T 与 in T …… 205
6.3.2 声明处型变 …… 206
6.3.3 类型投影 …… 206
6.4 泛型类 …… 207
6.5 泛型函数 …… 208
6.6 本章小结 …… 208
第 7 章 面向对象编程 …… 209
7.1 面向对象编程思想 …… 209
7.1.1 一切皆是映射 …… 209
7.1.2 二进制 01 与易经阴阳 …… 211
7.1.3 从面向过程到面向对象 …… 212
7.2 类与构造函数 …… 215
7.2.1 声明类 …… 216
7.2.2 构造函数 …… 216
7.2.3 类的属性 …… 218
7.2.4 函数（方法） …… 219
7.3 抽象类 …… 219
7.3.1 抽象类的定义 …… 219
7.3.2 抽象函数 …… 220
7.3.3 抽象属性和抽象类 …… 221
7.4 接口 …… 223
7.4.1 接口定义 …… 223
7.4.2 实现接口 …… 224
7.4.3 覆盖冲突 …… 225
7.4.4 接口中的属性 …… 225
7.5 抽象类和接口的差异 …… 226
7.6 继承 …… 227
7.6.1 open 类 …… 227
7.6.2 多重继承 …… 228
7.7 枚举类 …… 230
7.7.1 枚举类基本用法 …… 231
7.7.2 初始化枚举值 …… 231
7.7.3 使用枚举常量 …… 232
7.8 注解类 …… 233
7.8.1 声明注解 …… 233
7.8.2 使用注解 …… 234
7.9 单例模式与伴生对象 …… 234
7.9.1 单例模式 …… 234
7.9.2 object 对象 …… 235
7.9.3 嵌套 object 对象 …… 236
7.9.4 匿名 object …… 238
7.9.5 伴生对象 …… 239
7.10 密封类 …… 242
7.11 数据类 …… 243
7.11.1 构造函数中的 val/var …… 243
7.11.2 领域实体类 …… 245
7.11.3 数据类的概念 …… 245
7.11.4 数据类的限制 …… 246
7.11.5 数据类的解构 …… 248
7.11.6 标准数据类 Pair 和 Triple … 248
7.12 嵌套类 …… 249
7.12.1 嵌套类：类中的类 …… 249
7.12.2 内部类 …… 250

7.12.3 匿名内部类 …… 250
7.13 委托 …… 252
7.13.1 代理模式 …… 252
7.13.2 类的委托 …… 253
7.13.3 委托属性 …… 255
7.14 本章小结 …… 260

第 8 章 函数式编程 …… 261

8.1 函数式编程概述 …… 261
8.1.1 函数式编程与面向对象编程 …… 262
8.1.2 函数式编程基本特性 …… 265
8.1.3 函数式编程核心思想 …… 266
8.2 在 Kotlin 中使用函数式编程 …… 284
8.2.1 Kotlin 中的函数 …… 284
8.2.2 扩展函数 …… 286
8.2.3 中缀函数 …… 286
8.2.4 函数参数 …… 287
8.2.5 函数返回类型 …… 289
8.2.6 单表达式函数 …… 289
8.2.7 函数作用域 …… 289
8.2.8 泛型函数 …… 290
8.2.9 高阶函数 …… 290
8.2.10 匿名函数 …… 291
8.2.11 Lambda 表达式 …… 291
8.2.12 it：单个参数的隐式名称 …… 292
8.2.13 闭包 …… 292
8.2.14 带接收者的函数字面值 …… 293
8.2.15 具体化的类型参数 …… 293
8.2.16 尾递归 tailrec …… 294
8.3 本章小结 …… 295

第 9 章 轻量级线程：协程 …… 296

9.1 协程的基本概念 …… 296
9.2 协程的基本操作 …… 297
9.2.1 搭建项目 …… 297
9.2.2 简单协程示例 …… 298
9.2.3 使用 launch 函数启动协程 …… 299
9.2.4 CommonPool 对象 …… 299
9.2.5 挂起函数 …… 301
9.2.6 runBlocking 函数 …… 302
9.2.7 等待一个协程任务执行完毕 …… 303
9.2.8 取消协程 …… 306
9.2.9 协程取消失效 …… 307
9.2.10 计算代码协程的取消方式 …… 308
9.2.11 在 finally 中的协程代码 …… 310
9.2.12 协程中不被取消的代码块 …… 312
9.2.13 设置协程超时时间 …… 313
9.3 挂起函数 …… 314
9.3.1 按默认顺序执行 …… 314
9.3.2 使用异步并发执行 …… 315
9.4 协程上下文与调度器 …… 316
9.4.1 调度和线程 …… 316
9.4.2 父子协程 …… 318
9.5 通道与管道 …… 319
9.5.1 通道 vs 阻塞队列 …… 319
9.5.2 关闭通道和迭代遍历元素 …… 320
9.5.3 生产者 - 消费者模式 …… 321
9.5.4 使用管道生产无限序列 …… 322
9.5.5 管道与无穷质数序列 …… 323
9.5.6 通道缓冲区 …… 324
9.5.7 构建无穷惰性序列 …… 324
9.6 协程与线程 …… 325
9.6.1 轻量级的协程 …… 325
9.6.2 协程 vs 守护线程 …… 326
9.6.3 区别 …… 327
9.6.4 协程的好处 …… 328

9.6.5 Kotlin 协程的内部机制 …… 328
9.7 本章小结 …… 329

第 10 章 Kotlin 与 Java 互操作 …… 330

10.1 在 Kotlin 中调用 Java …… 330
10.1.1 调用 Okhttp 库示例 …… 331
10.1.2 使用 Java 的集合类 …… 332
10.1.3 调用 Java 中的 Getter 和 Setter …… 333
10.1.4 调用 Java 中返回 void 的方法 …… 334
10.1.5 调用来自 Java 中的 null …… 335
10.1.6 使用 Java 的泛型 …… 336
10.1.7 数组的互操作 …… 337
10.1.8 可变参数的互操作 …… 337
10.1.9 检查 Java 中的异常 …… 337
10.1.10 调用 Java 的 Object 方法 … 338
10.1.11 访问 Java 静态成员 …… 340
10.1.12 使用 Java 的反射 …… 340
10.1.13 SAM 转换 …… 341
10.2 在 Java 中调用 Kotlin …… 342
10.2.1 访问 Kotlin 属性 …… 342
10.2.2 调用 Kotlin 的包级函数 …… 344
10.2.3 调用 Kotlin 中的实例字段 … 347
10.2.4 调用 Kotlin 中的静态字段 … 348
10.2.5 调用 Kotlin 中的静态方法 … 349
10.2.6 Kotlin 与 Java 的可见性 …… 350
10.2.7 生成默认参数函数的重载 … 352
10.2.8 显式声明 Kotlin 中的异常 … 353
10.2.9 关键字冲突的互操作 …… 355
10.2.10 Nothing 类型 …… 355
10.3 类型映射 …… 356
10.4 Kotlin 与 Java 的简单实例对比 … 357
10.5 本章小结 …… 364

第三部分 综合实战

第 11 章 使用 Kotlin 集成 Spring Boot 开发 Web 服务端 …… 366

11.1 Spring Boot 简介 …… 366
11.2 系统架构技术栈 …… 367
11.3 准备工作 …… 367
11.3.1 创建工程 …… 367
11.3.2 Gradle 配置文件说明 …… 369
11.4 数据库层开发 …… 370
11.4.1 Endpoint 监控接口 …… 373
11.4.2 实体类 …… 375
11.4.3 数据访问层代码 …… 377
11.5 后端逻辑开发 …… 379
11.5.1 控制器逻辑层 …… 379
11.5.2 启动初始化 CommandLine-Runner …… 380
11.5.3 应用启动类 …… 380
11.5.4 Model 数据绑定 …… 382
11.5.5 模板引擎视图页面 …… 383
11.5.6 运行测试 …… 383
11.6 前端后端集成 …… 384
11.6.1 引入前端组件 …… 384
11.6.2 实现写文章模块 …… 386
11.6.3 加上导航栏 …… 388
11.6.4 抽取公共模板文件 …… 389
11.6.5 写文章的控制器层接口 …… 390
11.6.6 前端 Ajax 请求 …… 390
11.6.7 文章详情页 …… 391
11.6.8 添加 Markdown 支持 …… 393
11.6.9 文章列表分页搜索 …… 395

11.7 Spring 5.0 对 Kotlin 的支持 …… 397
11.8 本章小结 …… 399

第 12 章 使用 Kotlin 集成 Gradle 开发 …… 400

12.1 使用 Gradle 构建 Kotlin 工程 …… 400
12.2 使用 Kotlin 编写 Gradle 构建配置文件 …… 405
12.3 本章小结 …… 407

第 13 章 使用 Kotlin 和 Anko 的 Android 开发 …… 408

13.1 什么是 Anko …… 408
13.2 快速入门实例 …… 409
13.2.1 功能介绍 …… 410
13.2.2 使用 Android Studio 新建项目 …… 411
13.2.3 设计 UI 界面主题颜色 …… 415
13.2.4 核心依赖配置 …… 417
13.2.5 代码编写 …… 421
13.2.6 运行测试 …… 435
13.3 本章小结 …… 437

第 14 章 使用 Kotlin DSL …… 438

14.1 什么是 DSL …… 439
14.2 Kotlin 的 DSL 特性支持 …… 441
14.3 使用 kotlinx.html DSL 写前端代码 …… 442
14.4 实现一个极简的 HTTP DSL …… 447
14.5 本章小结 …… 454

第 15 章 Kotlin 文件 IO 操作与多线程 …… 455

15.1 Kotlin IO 基本操作 …… 455
15.1.1 终端 IO 操作 …… 455
15.1.2 读文件 …… 456
15.1.3 写文件 …… 457
15.1.4 遍历文件树 …… 458
15.1.5 网络 IO 操作 …… 459
15.1.6 kotlin.io 标准库 …… 460
15.1.7 执行 Shell 命令行 …… 461
15.2 正则表达式 …… 463
15.2.1 构造 Regex 表达式 …… 463
15.2.2 Regex 类中的函数 …… 463
15.2.3 使用 Java 正则表达式类 …… 465
15.3 Kotlin 的多线程 …… 466
15.3.1 创建线程 …… 466
15.3.2 同步方法和块 …… 467
15.3.3 可变字段 …… 468
15.4 本章小结 …… 469

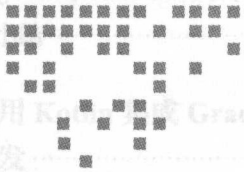
第 16 章 使用 Kotlin Native …… 470

16.1 Kotlin Native 简介 …… 470
16.2 快速开始 Hello World …… 471
16.3 Kotlin Native 与 C 语言互操作 …… 477
16.4 Kotlin Native 编译器 konan 简介 …… 486
16.5 本章小结 …… 494

第一部分 Part 1

Kotlin 语言基础

- 第 1 章　Kotlin 简介
- 第 2 章　快速开始 HelloWorld
- 第 3 章　Kotlin 语言基础
- 第 4 章　基本数据类型与类型系统



Chapter 1 第 1 章

Kotlin 简介

科特林岛（Котлин）是一座俄罗斯的岛屿，位于圣彼得堡以西约 30 公里处，形状狭长，东西长度约 14 公里，南北宽度约 2 公里，面积有 16 平方公里，扼守俄国进入芬兰湾的水道。科特林岛上建有喀琅施塔得市，为圣彼得堡下辖的城市。我们这里讲的 Kotlin 就是一门以这个岛命名的现代程序设计语言。本章将介绍 Kotlin 简史和快速学习工具。

1.1 Kotlin 简史



Kotlin 是一门静态类型编程语言，支持 JVM 平台、Android 平台、浏览器 JS 运行环境、本地机器码等。支持与 Java、Android 100% 完全互操作。

其主要设计者是来自 Saint Petersburg，Russia JetBrains 团队的布雷斯拉夫（Andrey Breslav，https://www.linkedin.com/in/abreslav/）等人，源码在 GitHub 上，其实现者主要是 JetBrains 团队成员以及开源贡献者。

认识一个事物的最好的方式，首先是去了解它的历史。我们先来简单看一下 Kotlin 的简历，见表 1-1。

表 1-1 Kotlin 的简历

标　题	内　容
设计者	JetBrains
实现者	JetBrains 与开源贡献者
最新发行时间	Kotlin 1.1.2（2017 年 4 月 25 日）
最新测试版发行日期	Kotlin 1.1.3 EAP（2017 年 5 月 27 日）
类型系统	静态类型

（续）

标　题	内　容
系统平台	输出 Java 虚拟机比特码以及 JavaScript 源代码
操作系统	任何支持 JVM 或者 JavaScript 的解释器
许可证	Apache 2
文件扩展名	.kt
网站	kotlinlang.org
启发语言	Java、Scala、Groovy、C#、Gosu

Kotlin 的亲爹是大名鼎鼎的 JetBrains 公司。该公司有一系列耳熟能详的产品，诸如 Android 程序员们天天用的 Android Studio、IntelliJ IDEA，还有前端的 WebStorm、PhpStorm 等。所以使用 IntelliJ IDEA 开发 Kotlin 程序将会非常便捷。

1.1.1　Kotlin 概述

Kotlin 语言从一开始推出到如今已经有六年了，但官方正式发布首个稳定版本的时间相对比较晚（2016 年 2 月），其大致发展简史如下：

- 2011 年 7 月，JetBrains 推出 Kotlin 项目。
- 2012 年 2 月，JetBrains 以 Apache 2 许可证开源此项目。
- 2016 年 2 月 15 日，Kotlin v1.0（第一个官方稳定版本）发布。
- 2017 Google I/O 大会，Kotlin “转正”。

Kotlin 具有很多下一代编程语言静态语言特性：如类型推断、多范式支持、可空性表达、扩展函数、模式匹配等。

Kotlin 的编译器 kompiler 可以被独立出来并嵌入到 Maven、Ant 或 Gradle 工具链中。这使得在 IDE 中开发的代码能够利用已有的机制来构建，可以在新环境中自由使用。

1.1.2　Kotlin 元年：2016

2016 年是 Kotlin 的元年，官网给出了这样一幅图来展示它一年来的成绩，如图 1-1 所示。



图 1-1　Kotlin 在 2016 年的成绩

从图中可见，GitHub 上面的代码量破千万，有 8000 多个基于 Kotlin 项目，Kotlin 的用户也在逐渐增多。

Kotlin 是由工程师设计，各种细节设计非常契合工程师的需要。语法近似于 Java 和 Scala，且已活跃在 Android 开发领域，被誉为 Android 平台的 Swift。

其主要设计目标如下：

- 创建一种兼容 Java 的语言。
- 比 Java 更安全，能够静态检测常见的陷阱。如：引用空指针。
- 比 Java 更简洁，通过支持 variable type inference、higher-order functions（closures）、extension functions、mixins、first-class delegation 等实现。
- 比最成熟的竞争对手 Scala 语言更加简单。

Kotlin 的学习曲线极其平缓，学习量相当于掌握一个框架，有经验的程序员阅读了文档即可上手。

1.2 快速学习工具

工欲善其事必先利其器。要快速学习 Kotlin，先要了解其开发环境搭建。下面分别介绍。

1.2.1 云端 IDE

未来的是云的世界。不需要搭建本地开发运行环境，直接用浏览器打开：https://try.kotlinlang.org/

你就可以直接使用云端 IDE 来即时编写 Kotlin 代码，并运行之。一个运行示例如图 1-2 所示。

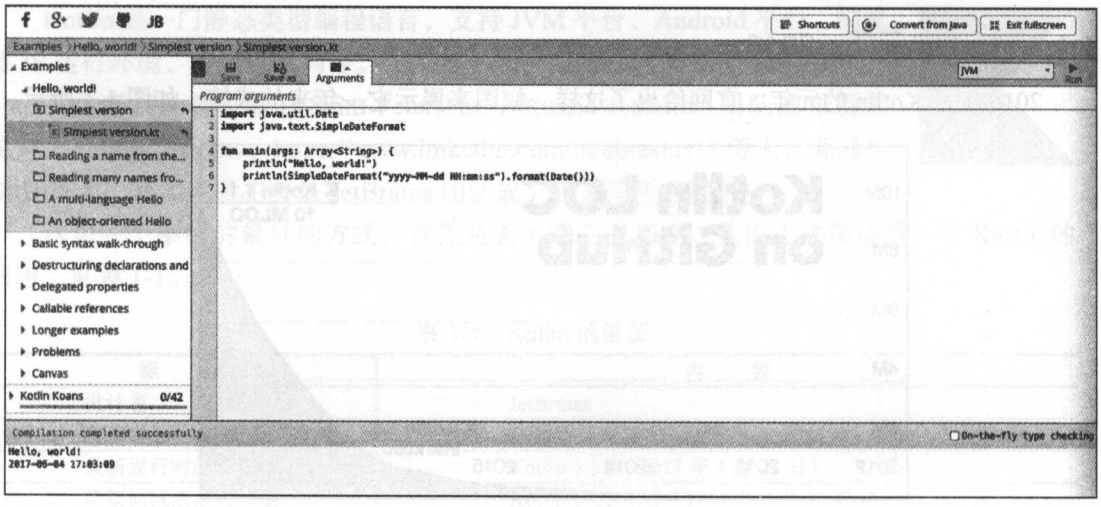


图 1-2 云端 IDE 示例

1.2.2　本地命令行环境搭建

Kotlin 是运行在 JVM 环境下的语言。首先我们要有 JDK 环境。

有时候我们并不需要打开 IDE 来做一些事情，因为打开 IDE 是件很麻烦的事情。在某些场景下，我们比较喜欢命令行。

使用命令行环境，我们可以方便地使用 Kotlin REPL（Read-Eval-Print-Loop，交互式编程环境）。用 REPL 可以实时编写 Kotlin 代码，并查看运行结果。通常 REPL 交互方式可以用于调试、测试以及试验某种想法。

下面我们介绍如何搭建 Kotlin 命令行环境。Kotlin 命令行环境主要依赖于 Kotlin Compiler，目前最新版本是 1.1.2-2。其下载链接是：https://github.com/JetBrains/kotlin/releases/tag/v1.1.2-2。

这个 zip 包里面就是 Kotlin Compiler 的核心依赖 jar 包。解压后，目录结构如下：

```
.
├── bin
│   ├── kotlin
│   ├── kotlin.bat
│   ├── kotlinc
│   ├── kotlinc-js
│   ├── kotlinc-js.bat
│   ├── kotlinc-jvm
│   ├── kotlinc-jvm.bat
│   └── kotlinc.bat
├── build.txt
├── lib
│   ├── allopen-compiler-plugin.jar
│   ├── android-extensions-compiler.jar
│   ├── kotlin-annotation-processing.jar
│   ├── kotlin-ant.jar
│   ├── kotlin-build-common-test.jar
│   ├── kotlin-compiler-client-embeddable.jar
│   ├── kotlin-compiler.jar
│   ├── kotlin-daemon-client.jar
│   ├── kotlin-jslib-sources.jar
│   ├── kotlin-jslib.jar
│   ├── kotlin-preloader.jar
│   ├── kotlin-reflect.jar
│   ├── kotlin-runner.jar
│   ├── kotlin-runtime-sources.jar
│   ├── kotlin-runtime.jar
│   ├── kotlin-script-runtime-sources.jar
│   ├── kotlin-script-runtime.jar
│   ├── kotlin-stdlib-js-sources.jar
│   ├── kotlin-stdlib-js.jar
│   ├── kotlin-stdlib-sources.jar
│   ├── kotlin-stdlib.jar
```

```
|   ├── kotlin-test-js.jar
|   ├── kotlin-test.jar
|   ├── noarg-compiler-plugin.jar
|   ├── sam-with-receiver-compiler-plugin.jar
|   └── source-sections-compiler-plugin.jar
└── license
    ├── LICENSE.txt
    ├── NOTICE.txt
    └── third_party
        ├── args4j_LICENSE.txt
        ├── asm_license.txt
        ├── closure-compiler_LICENSE.txt
        ├── dart_LICENSE.txt
        ├── jshashtable_license.txt
        ├── json_LICENSE.txt
        ├── maven_LICENSE.txt
        ├── pcollections_LICENSE.txt
        ├── prototype_license.txt
        ├── rhino_LICENSE.txt
        ├── scala_license.txt
        ├── trove_license.txt
        └── trove_readme_license.txt

4 directories, 50 files
```

其中，kotlinc 和 kotlin 两个命令就是 Kotlin 语言的编译 .kt 文件和运行 Kt.class 文件命令。我们来看一下 kotlinc 命令的 shell 脚本，如下所示：

```
#!/usr/bin/env bash
#
##############################################################################
# Copyright 2002-2011, LAMP/EPFL
# Copyright 2011-2015, JetBrains
#
# This is free software; see the distribution for copying conditions.
# There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
# PARTICULAR PURPOSE.
##############################################################################

cygwin=false;
case "`uname`" in
    CYGWIN*) cygwin=true ;;
esac

# Based on findScalaHome() from scalac script
findKotlinHome() {
    local source="${BASH_SOURCE[0]}"
    while [ -h "$source" ] ; do
        local linked="$(readlink "$source")"
        local dir="$(cd -P $(dirname "$source") && cd -P $(dirname "$linked") && pwd)"
```

```
        source="$dir/$(basename "$linked")"
    done
    (cd -P "$(dirname "$source")/.." && pwd)
}

KOTLIN_HOME="$(findKotlinHome)"

if $cygwin; then
    # Remove spaces from KOTLIN_HOME on windows
    KOTLIN_HOME=`cygpath --windows --short-name "$KOTLIN_HOME"`
fi

[ -n "$JAVA_OPTS" ] || JAVA_OPTS="-Xmx256M -Xms32M"

declare -a java_args
declare -a kotlin_args

while [ $# -gt 0 ]; do
    case "$1" in
        -D*)
            java_args=("${java_args[@]}" "$1")
            shift
            ;;
        -J*)
            java_args=("${java_args[@]}" "${1:2}")
            shift
            ;;
        *)
            kotlin_args=("${kotlin_args[@]}" "$1")
            shift
            ;;
    esac
done

if [ -z "$JAVACMD" -a -n "$JAVA_HOME" -a -x "$JAVA_HOME/bin/java" ]; then
    JAVACMD="$JAVA_HOME/bin/java"
fi

declare -a kotlin_app

if [ -n "$KOTLIN_RUNNER" ];
then
    java_args=("${java_args[@]}" "-Dkotlin.home=${KOTLIN_HOME}")
    kotlin_app=("${KOTLIN_HOME}/lib/kotlin-runner.jar" "org.jetbrains.kotlin.
        runner.Main")
else
    [ -n "$KOTLIN_COMPILER" ] || KOTLIN_COMPILER=org.jetbrains.kotlin.cli.jvm.
        K2JVMCompiler
    java_args=("${java_args[@]}" "-noverify")
    kotlin_app=("${KOTLIN_HOME}/lib/kotlin-preloader.jar" "org.jetbrains.kotlin.
```

```
        preloading.Preloader" "-cp" "${KOTLIN_HOME}/lib/kotlin-compiler.jar"
        $KOTLIN_COMPILER)
fi

"${JAVACMD:=java}" $JAVA_OPTS "${java_args[@]}" -cp "${kotlin_app[@]}" "${kotlin_args[@]}"
```

可以看出，kotlinc 是直接依赖 Java 命令的，所以，使用 Kotlin Compiler，首先要有 JDK 环境。其中 kotlin-preloader.jar、kotlin-compiler.jar 是其入口依赖 jar，入口类是 org.jetbrains.kotlin.cli.jvm.K2JVMCompiler。

kotlin 命令脚本如下：

```
export KOTLIN_RUNNER=1

DIR="${BASH_SOURCE[0]%/*}"
: ${DIR:="."}

"${DIR}"/kotlinc "$@"
```

我们可以看出，直接依赖 kotlinc。在 if 逻辑代码中，我们可以看出，Kt.class 在 Java 命令执行前，需要从 kotlin-runner.jar 这个逻辑里走一遍。

```
if [ -n "$KOTLIN_RUNNER" ];
then
    java_args=("${java_args[@]}" "-Dkotlin.home=${KOTLIN_HOME}")
    kotlin_app=("${KOTLIN_HOME}/lib/kotlin-runner.jar" "org.jetbrains.kotlin.
        runner.Main")
```

同时，我们也能知道 Kt.class 跟 Java.class 文件有着这个 kotlin-runner.jar 的逻辑映射上的区别。也就是说，Kotlin 的 bytecode 跟纯的 JVM bytecode 存在一个 kotlin-runner.jar 的映射关系。其大致执行过程，如图 1-3 所示。

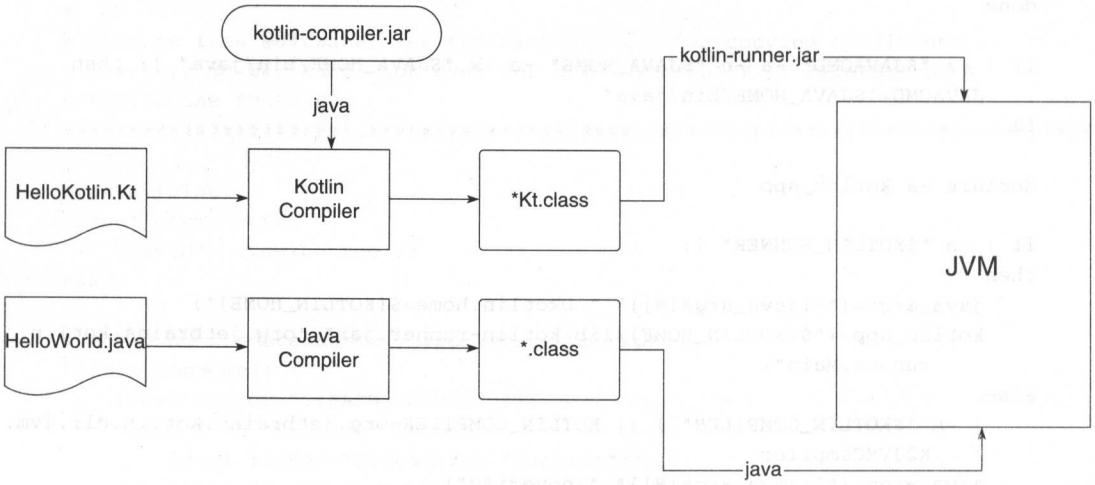


图 1-3 Kotlin 的 Bytecode 执行过程

像Scala、Groovy等基于JVM的语言，其compiler、runner基本都采用这种运行方式。在实现细节上也许会有不同，总的思路是一致的。比如说，scalac的入口类如下：

```
https://github.com/EasyKotlin/scala/blob/2.12.x/src/compiler/scala/tools/nsc/Main.scala
```

对应scalac中的命令行脚本是：

```
...

execCommand \
    "${JAVACMD:=java}" \
    $JAVA_OPTS \
    "${java_args[@]}" \
    "${classpath_args[@]}" \
    -Dscala.home="$SCALA_HOME" \
    $OVERRIDE_USEJAVACP \
    "$EMACS_OPT" \
    $WINDOWS_OPT \
     scala.tools.nsc.Main  "$@"

     ...
```

我们解压完kotlin-compiler-1.1.2-2.zip，放到相应的目录下。然后配置系统环境变量：

```
export KOTLIN_HOME=/Users/jack/soft/kotlinc
export PATH=$PATH:$KOTLIN_HOME/bin
```

执行source ~/.bashrc，命令行输入kotlinc，即可进入REPL环境，我们可以看到如下输出：

```
$ kotlinc
Welcome to Kotlin version 1.1.2-2 (JRE 1.8.0_40-b27)
Type :help for help, :quit for quit
>>> println("Hello,World")
Hello,World
>>>
```

然后，我们就可以像使用Python、Ruby、Scala、Groovy的REPL一样去尽情享受Kotlin的编程乐趣了。

1.2.3 使用IntelliJ IDEA

最新版本的IDEA已经默认集成了Kotlin环境。

首先去下载安装IntelliJ IDEA。下载地址是：

```
https://www.jetbrains.com/idea/download/index.html
```

如果之前没用过IDEA，现在想尝试一下，可以去下面这个页面了解一下：

```
https://www.jetbrains.com/idea/documentation/
```

安装完毕，然后点击File → New → Project，再选择：

```
Koltlin: Kotlin(JVM), Kotlin(JavaScript)
```

出现如图 1-4 所示的界面。

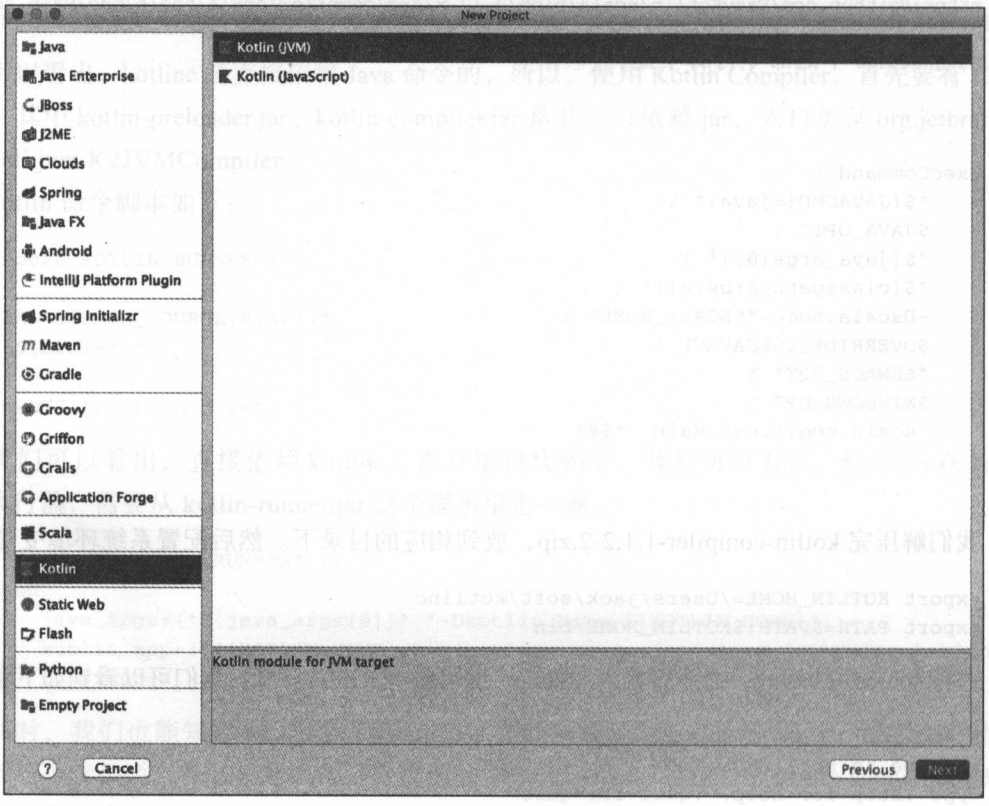


图 1-4 使用 IntelliJ IDEA

也可以选择 Maven、Gradle 构建项目。本书采用 Gradle 来构建项目，如图 1-5 所示。然后按照后续步骤操作，最后等待 Gradle 下载依赖，完成项目构建。我们将得到一个标准的 Gradle 项目，如图 1-6 所示。

我们在 src/main/kotlin 下面新建 package：com.easy.kotlin.chaptor1。

然后新建 HelloWorld.kt，编写以下代码：

```
package com.easy.kotlin.chaptor1

/**
 * Created by jack on 2017/6/5.
 */

fun main(args:Array<String>){
    println("Hello,World!")
}
```

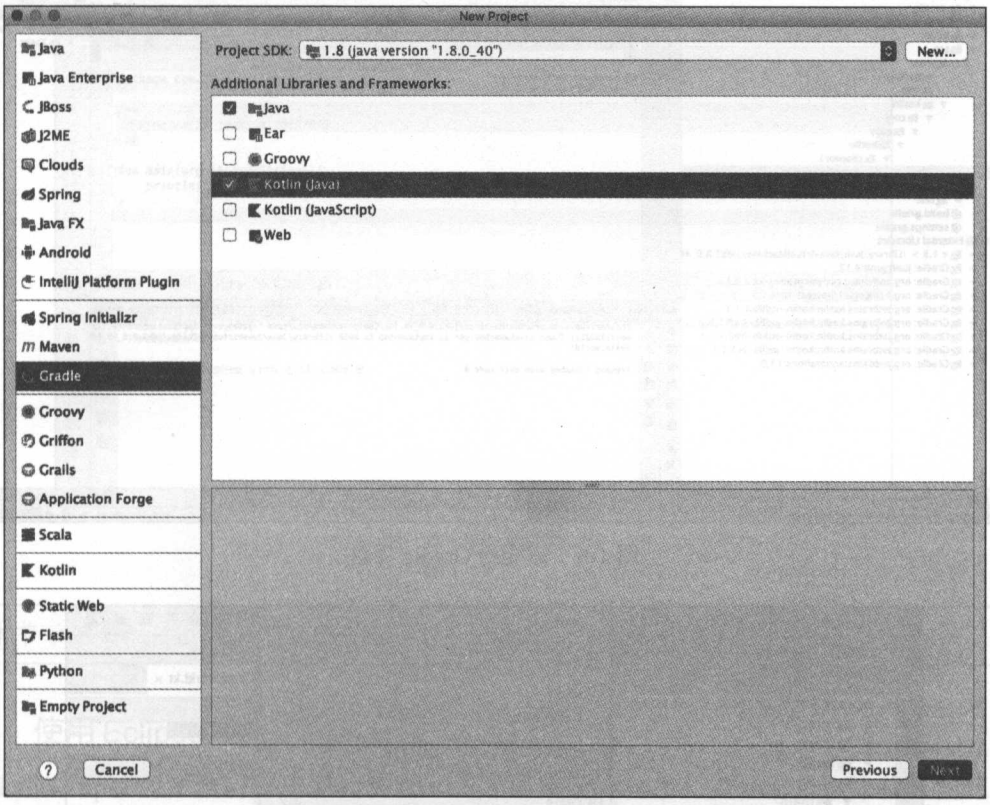


图 1-5　用 Gradle 来构建项目

右击运行该类，如图 1-7 所示。

我们将会得到输出，如图 1-8 所示。

观察 IDEA 控制台输出的执行日志，可以看出 IDEA 集成 Kotlin 环境使用的核心依赖 jar 包：

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/bin/java "-javaagent:/
    Applications/IntelliJ IDEA.app/Contents/lib/idea_rt.jar=65404:/Applications/
    IntelliJ IDEA.app/Contents/bin" -Dfile.encoding=UTF-8 -classpath /Library/
    Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/jre/lib/charsets.jar:/
    Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/jre/lib/deploy.
    jar:...
...

.../kotlin-stdlib-jre8-1.1.1.jar:
.../kotlin-stdlib-jre7-1.1.1.jar:
.../kotlin-stdlib-1.1.1.jar:... com.easy.kotlin.chaptor1.HelloWorldKt
...


Hello,World!

Process finished with exit code 0
```

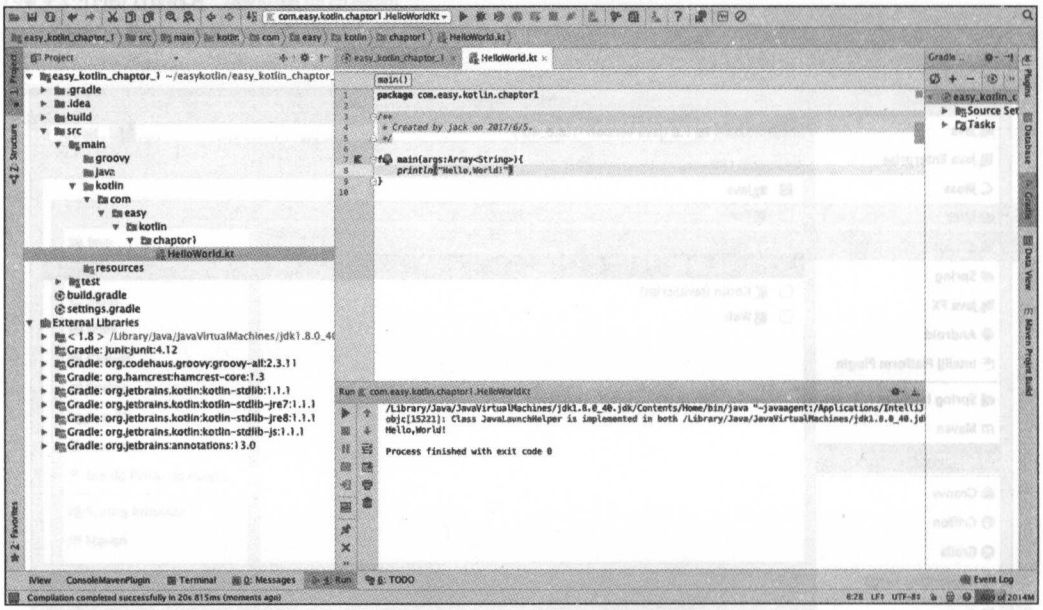


图 1-6　标准的 Gradle 项目

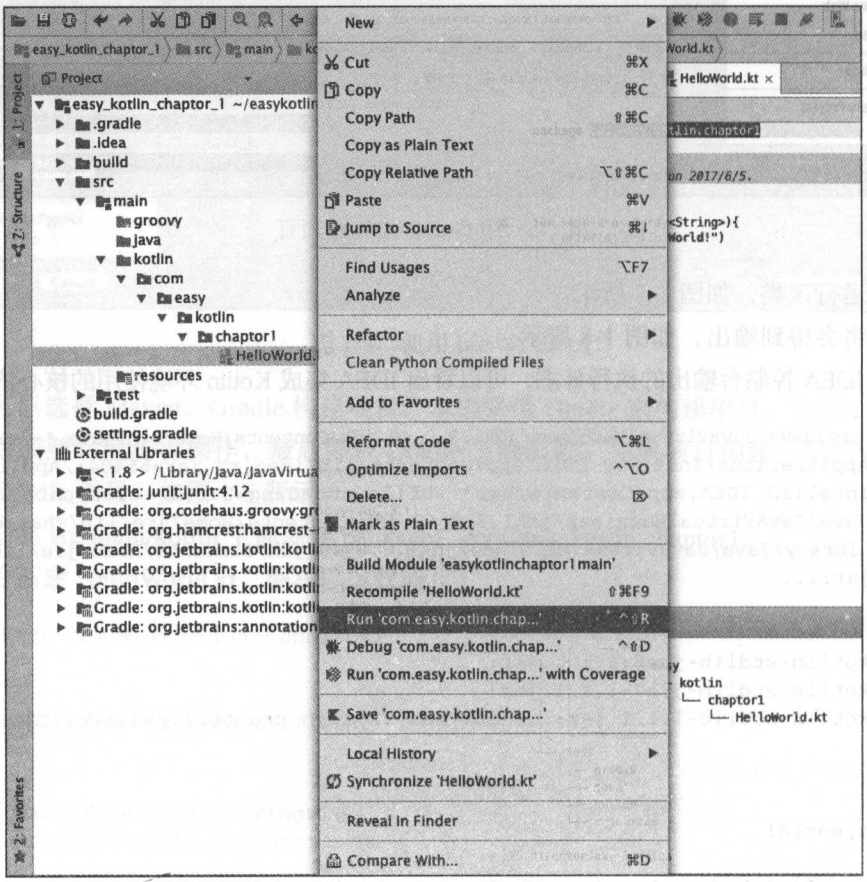


图 1-7　右击运行该类

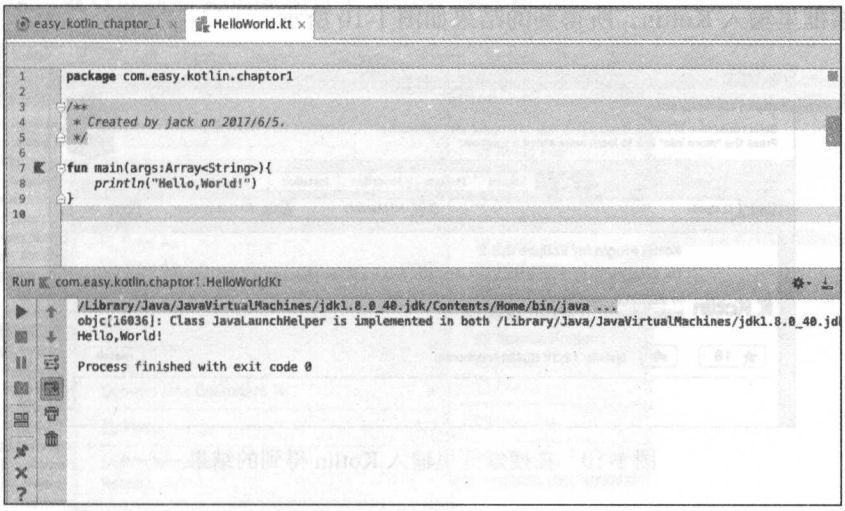


图 1-8　输出结果

本节示例代码：https://github.com/EasyKotlin/easy_kotlin_chapter_1

1.2.4　使用 Eclipse

使用 Eclipse 的开发者可以通过安装 Kotlin 插件进行程序开发。但是，体验上要比使用 IDEA 逊色很多。如果想完美体验 Kotlin 在 IDE 中的开发，强烈建议使用 IDEA，因为 JetBrains 自家的东西自然是比 Eclipse 支持得要好很多。

下面简单介绍一下在 Eclipse 中开发 Kotlin 程序的方法。

首先，打开 Help → Eclipse Marketplace，如图 1-9 所示。

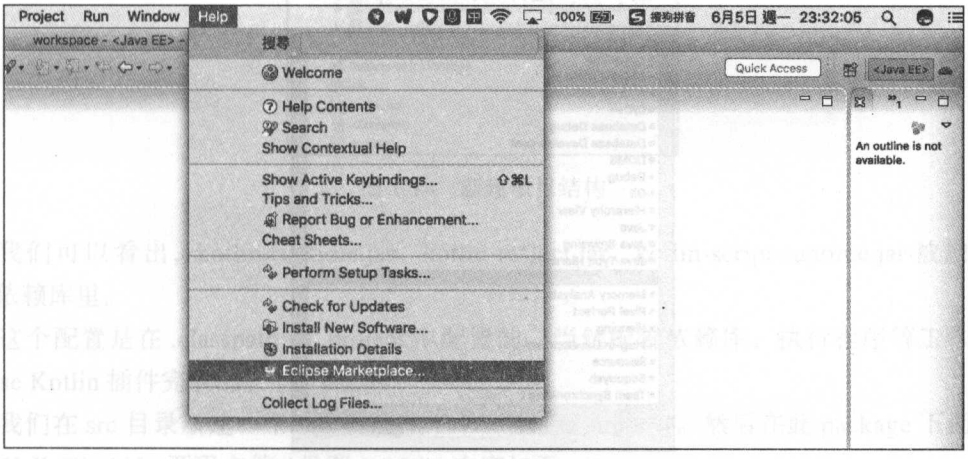


图 1-9　打开 Eclipse Marketplace

在搜索框里输入 Kotlin，所得到的结果如图 1-10 所示。

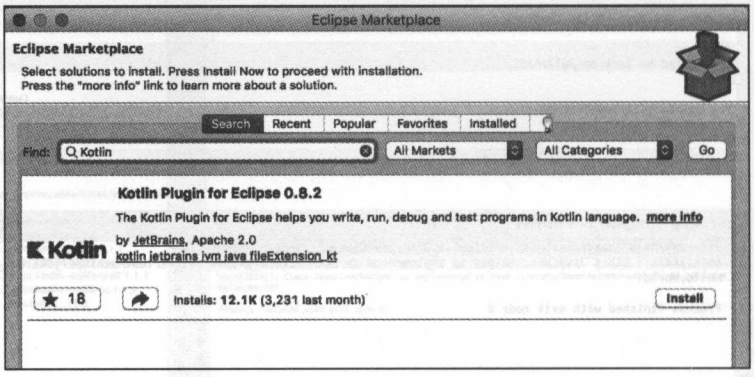


图 1-10 在搜索框里输入 Kotlin 得到的结果

点击 Install，等待完成安装，重启 Eclipse。

然后，选择 Kotlin Perspective，如图 1-11 所示，再选择 Kotlin，如图 1-12 所示。

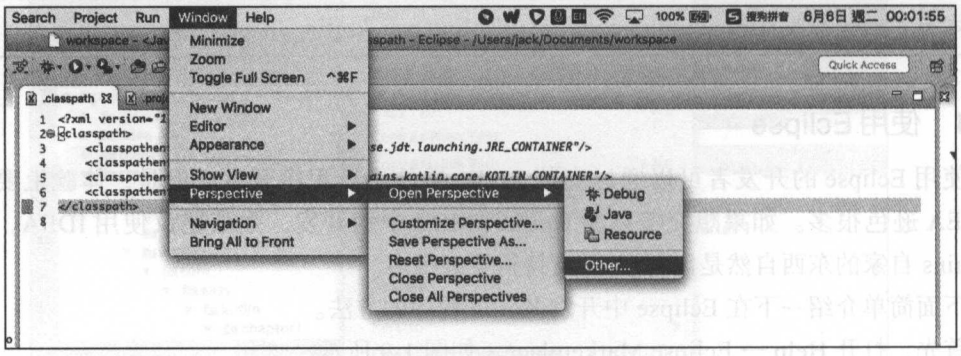


图 1-11 选择 Kotlin Perspective

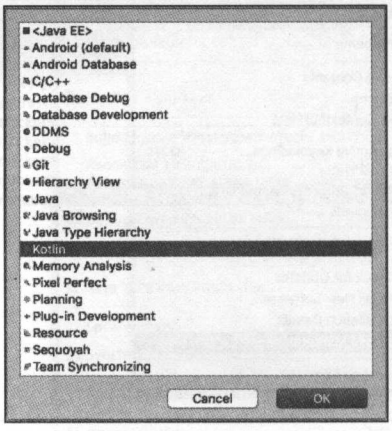


图 1-12 选择 Kotlin

点击 OK，就可以新建 Kotlin 项目了，如图 1-13 所示。

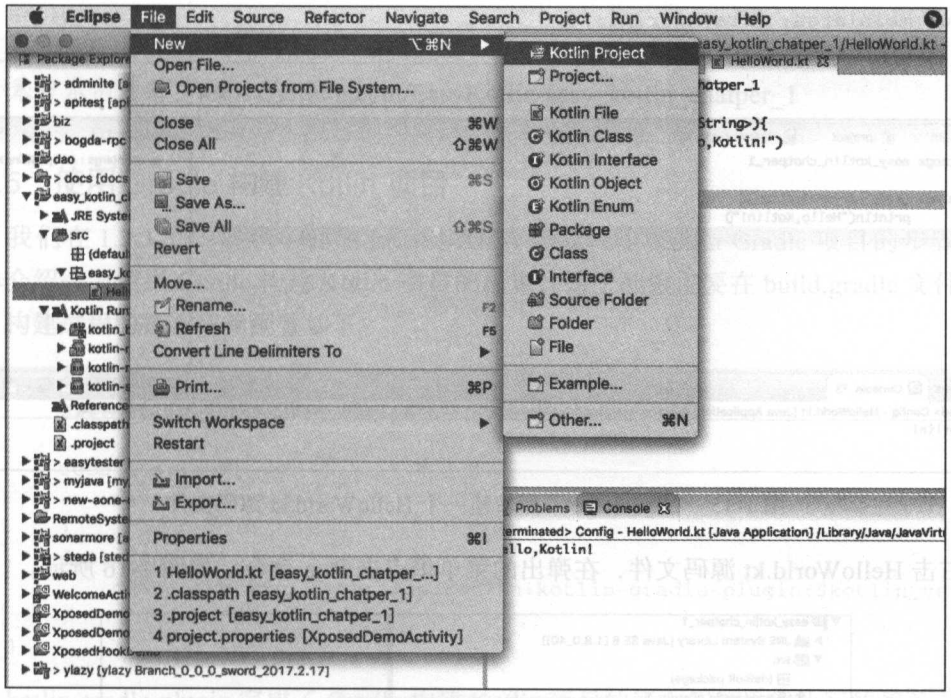


图 1-13　新建 Kotlin 项目

新建完项目，我们将得到如图 1-14 所示的项目结构。

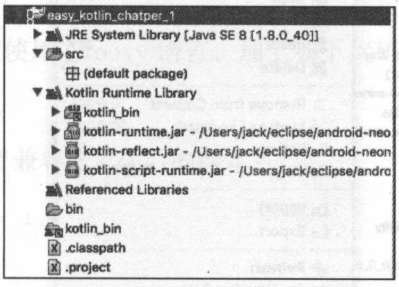


图 1-14　新建项目结构

我们可以看出，kotlin-runtime.jar、kotlin-reflect.jar、kotlin-script-runtime.jar 被加到了项目依赖库里。

这个配置是在 .classpath 和 .project 中配置的。当然配置依赖库、执行程序等工作是由 Eclipse Kotlin 插件完成的。

我们在 src 目录新建一个 package ： easy_kotlin_chatper_1。然后在此 package 下面新建一个 HelloWorld.kt 源码文件（见图 1-15），内容如下：

```
package easy_kotlin_chatper_1

fun main(args: Array<String>){
    println("Hello,Kotlin!")
}
```

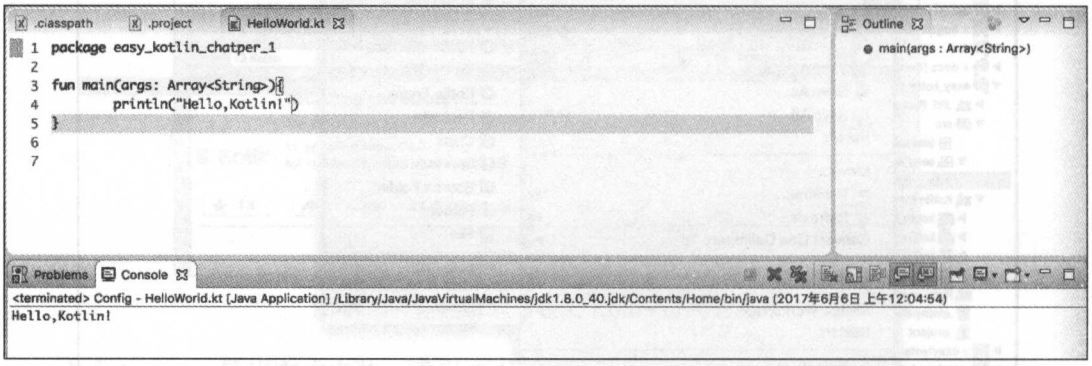


图 1-15 在 package 下面新建一个 HelloWorld.kt 源码文件

右击 HelloWorld.kt 源码文件，在弹出的菜单里点击 Run 命令，如图 1-16 所示。

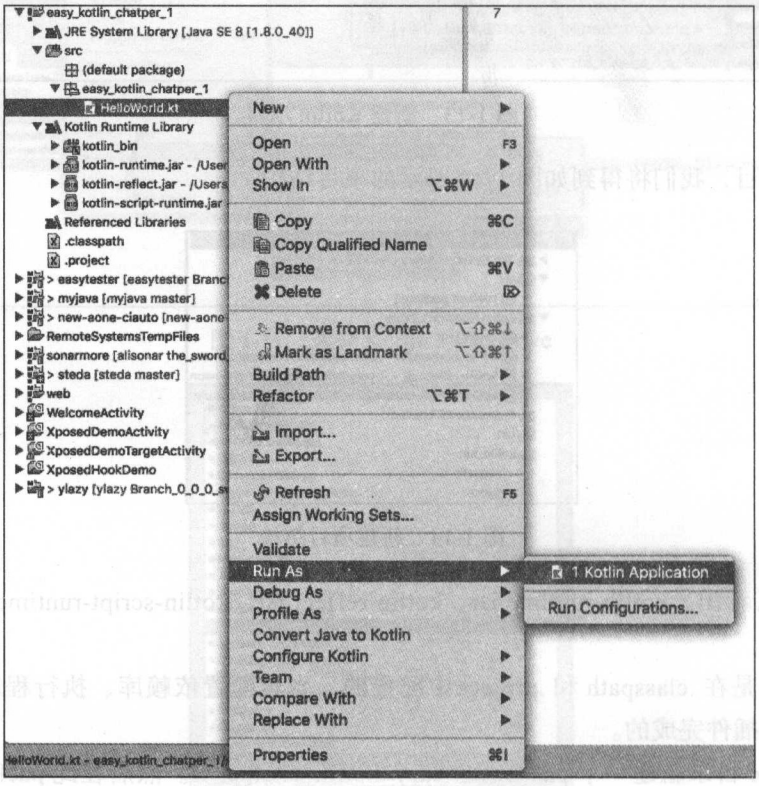


图 1-16 右击 HelloWorld.kt 源码文件并运行

如果一切正常，我们将得到如下输出：

```
Hello,Kotlin!
```

本节示例源码：https://github.com/EasyKotlin/easy_kotlin_chatper_1

1.2.5　使用 Gradle 构建 Kotlin 项目

我们在 1.2.3 节中展示了使用 IntelliJ IDEA 建立一个 Kotlin Gradle 项目的步骤，本节简单介绍一下使用 Gradle 构建 Kotlin 项目的配置。这个配置主要在 build.gradle 文件中，其中，构建过程的核心依赖配置如下：

```
buildscript {
    ext.kotlin_version = '1.1.1'

    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

kotlin-gradle-plugin 完成了 Gradle 构建 Kotlin 项目的所有依赖构建执行的相关工作。

然后，使用 Gradle java、kotlin 插件：

```
apply plugin: 'java'
apply plugin: 'kotlin'
```

当然，如果我们同时想使用 Groovy 语言，加上如下代码即可：

```
apply plugin: 'groovy'
```

源代码 JDK 兼容性配置兼容 1.8 往后的版本：

```
sourceCompatibility = 1.8
```

配置 Maven 仓库：

```
repositories {
    mavenCentral()
}
```

项目依赖：

```
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre8:$kotlin_version"
    compile "org.jetbrains.kotlin:kotlin-stdlib-js:$kotlin_version"
    compile 'org.codehaus.groovy:groovy-all:2.3.11'
```

```
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

其中，kotlin-stdlib-jre8 是 Kotlin JVM 执行环境依赖。org.jetbrains.kotlin:kotlin-stdlib-js 是 Kotlin JS 执行环境依赖。我们可以通过 Gradle 项目的依赖树（见图 1-17）看出 kotlin-stdlib-jre8 依赖：

```
org.jetbrains.kotlin:kotlin-stdlib:1.1.1
```

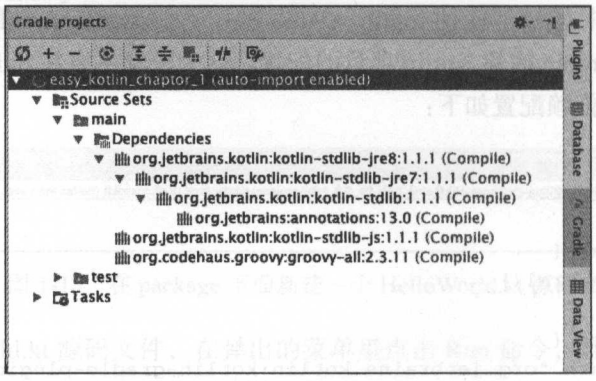


图 1-17 Gradle 项目的依赖树

kotlin-stdlib 是 Kotlin 运行环境的标准库。

1.2.6 编程语言学习小结

学习一门语言大概会经历如下四步。

1. 基本语法

学习任何东西，都是一个由表及里的过程，学习一门编程语言也一样。对于一门编程语言来说，“表”就是基本词汇和语法。对于基础语法的学习，我们可以看一些简短而又系统的教程。

2. 编码实践

所谓“纸上得来终觉浅，绝知此事要躬行”是也，此处就不多说了，总之，要掌握基础，持续练习。

每一门编程语言的学习内容都会涉及以下基本内容：

- 运行环境
- 数据类型（数字、字符串、数组、集合、映射字典等）
- 表达式
- 流程控制
- 类、方法与函数

不同的语言还有一些不同的特性，可以通过对比学习来加深理解，并通过大量实践深入理解，达到熟练使用。后面还要再去深入了解面向对象编程（OOP）、函数式编程（FP）、并发、异常、文件 IO、网络、标准库等内容，并辅以持续的练习，这些内容才能够让你真正进入编程领域并做出实际的软件。

相信 10000 小时定律，“No Pain，No Gain”。

学习一门新语言的时候，要利用以前所学的语言功底，但是也要保持开放的心态。这些在认知心理学中有相应的讲述。感兴趣的读者可以去了解一下。

3. 技近乎道

学习基础语法能让你快速上手，应用实践。对技巧和陷阱的关注，一定程度上拓展了你的知识面。系统学习一方面会进一步拓展你的知识面；另一方面，也有利于形成语言知识结构。

任何一门成熟语言，都有其特有的生态。这个生态包括：框架、扩展包、解决方案、模式、规范等。

在不断编码实践过程中，我们逐步熟练使用很多 API 库、框架，也不断踩坑填坑、看源代码，不断解决问题，不断加深对语言的理解，同时会看一些优秀的框架源代码。

如果还有精力，我们再去学习语言更底层的东西，如 Java 中的集合类实现的算法与数据结构，JVM 是如何执行 Java 代码的，Java 的线程和操作系统线程的关系，以及一些操作系统方面的知识，而不仅仅停留在应用层面。

最后，达到游刃有余的境界。进入这一层级，基本可入武林高手之列了。

《庖丁解牛》中说到：“臣之所好者，道也，进乎技矣。始臣之解牛之时，所见无非牛者。三年之后，未尝见全牛也。方今之时，臣以神遇而不以目视，官知止而神欲行。依乎天理，批大郤，导大窾，因其固然，技经肯綮之未尝，而况大軱乎！”可以将这里的“牛”理解为我们所说的各种编程思想、编程范式、编程方法、编程技巧等。最后，达到“运用之妙，存乎一心”之境也。

4. 创造新世界

“编程的本质就是创造世界。”达到这个境界的，基本都是世界顶尖大牛了。

编程语言发展史上的杰出人物（下面只是一份不完整名单）：

- 约翰·冯·诺伊曼：操作系统概念的发起者。
- 肯·汤普逊和丹尼斯·里奇：发明了 C 和 Unix。
- 约翰·巴科斯：发明了 Fortran。
- 阿兰·库珀：开发了 Visual Basic。
- 詹姆斯·高斯林：开发了 Oak，即后来的 Java。
- 安德斯·海尔斯伯格：开发了 Turbo Pascal、Delphi 以及 C#。
- 葛丽丝·霍普：开发了 Flow-Matic，该语言对 COBOL 造成了影响。

- 肯尼斯·艾佛森：开发了 APL，并与 Roger Hui 合作开发了 J。
- 比尔·乔伊：发明了 vi，BSD、Unix 的前期作者，也是 SunOS 的发起人，该操作系统后来改名为 Solaris。
- 艾伦·凯：开创了面向对象编程语言，也是 Smalltalk 的发起人。
- 布莱恩·克尼汉：与丹尼斯·里奇合著第一本 C 程序设计语言的书籍，同时也是 AWK 与 AMPL 程序设计语言的共同作者。
- 约翰·麦卡锡：发明了 LISP。
- 比雅尼·斯特劳斯特鲁普：开发了 C++。
- 尼克劳斯·维尔特：发明了 Pascal 与 Modula。
- 拉里·沃尔：创造了 Perl 与 Perl 6。
- 吉多·范罗苏姆：创造了 Python。

……

这些人，都在创造一个美妙的思维逻辑之塔，创造了一个新世界。正是这些各个编程领域的引领者们，才使得这个世界更加美好。

1.3 本章小结

本章简单介绍了 Kotlin 语言的发展过程，以及 Kotlin 开发环境的搭建方法。简单总结了学习一门编程语言的基本过程，这本书基本是按照这个思路来组织架构的。

在下一章中，你将体验在多平台上使用 Kotlin 语言进行开发的过程。

感谢你的阅读！恭喜你已经正式开启 Kotlin 世界之旅，希望本书能引导你学习 Kotlin，哪怕是给你一点点启发，我也倍感欣慰。

第 2 章 Chapter 2

快速开始 HelloWorld

我们在本章将快速开始使用 Kotlin，分别使用命令行、应用程序、Web RESTful、Android、Kotlin JavaScript 等方式向世界问好。

2.1 命令行版 HelloWorld

Kotlin 提供了方便的命令行 REPL 环境，可以方便我们快速编译执行一段代码，在语言的初学阶段这非常有用。

安装配置完 Kotlin 命令行环境之后，直接在命令行输入 kotlinc，即可进入 Kotlin REPL 界面：

```
$ kotlinc
Welcome to Kotlin version 1.1.2-2 (JRE 1.8.0_40-b27)
Type :help for help, :quit for quit
>>> println("Hello,World!")
Hello,World!

>>> import java.util.Date
>>> Date()
Wed Jun 07 14:19:33 CST 2017
```

2.2 应用程序版 HelloWorld

如果想拥有学习 Kotlin 的相对较好的体验，建议使用 IDEA。毕竟 Kotlin 是 JetBrains

家族的亲儿子，跟 Intelli IDEA 是血浓于水啊。

我们使用 IDEA 新建 Gradle 项目，选择 Kotlin（Java）框架支持，如图 2-1 所示。

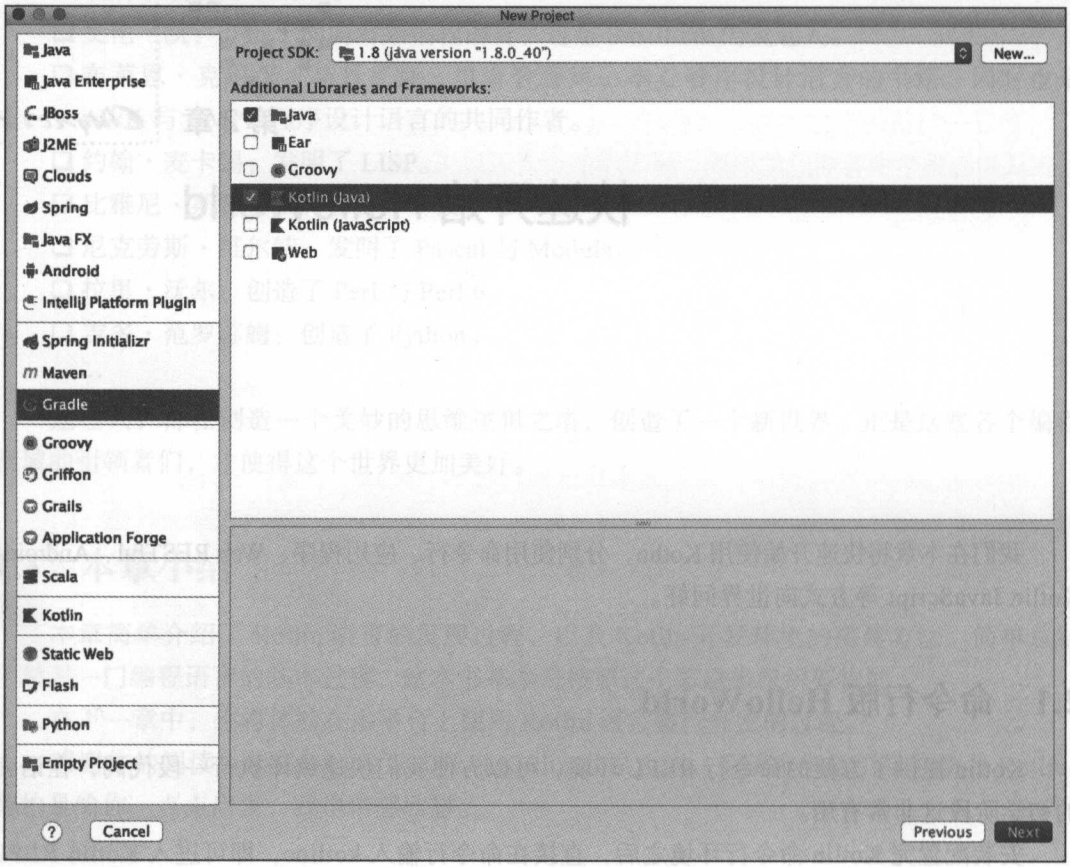


图 2-1 新建项目

新建完项目，我们写一个 HelloWorld.kt 类：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/5/29.
 */

import java.util.Date
import java.text.SimpleDateFormat

fun main(args: Array<String>) {
    println("Hello, world!")
    println(SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(Date()))
}
```

整体的项目目录结构如下：

```
.
├── README.md
├── build
│   ├── classes
│   │   └── main
│   │       ├── META-INF
│   │       │   └── easykotlin_main.kotlin_module
│   │       └── com
│   │           └── easy
│   │               └── kotlin
│   │                   └── HelloWorldKt.class
│   └── kotlin-build
│       └── caches
│           └── version.txt
├── build.gradle
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               └── HelloWorld.kt
    │   └── resources
    └── test
        ├── java
        ├── kotlin
        └── resources

21 directories, 7 files
```

直接运行 HelloWorld.kt，输出结果如下：

```
Hello, world!
2017-05-29 01:15:30
```

关于项目的编译、构建、运行，是由 Gradle 协同 kotlin-gradle-plugin 在 kotlin-stdlib-jre8、kotlin-stdlib 核心依赖下完成的。build.gradle 配置文件如下：

```
group 'com.easy.kotlin'
version '1.0-SNAPSHOT'

buildscript {
    ext.kotlin_version = '1.1.1'

    repositories {
        mavenCentral()
    }
```

```
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

apply plugin: 'java'
apply plugin: 'kotlin'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre8:$kotlin_version"
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

本节示例源码：https://github.com/EasyKotlin/easykotlin/tree/easykotlin_hello_world_20170529

2.3 Web RESTful HelloWorld

本节介绍使用 Kotlin 结合 Spring Boot 开发一个 RESTful 版本的 HelloWorld。

1. 新建 Gradle 的 Kotlin 项目

打开 IDEA 的 File → New → Project，如图 2-2 所示。

按照界面操作，输入相应的项目名等信息，即可新建一个使用 Gradle 构建的标准 Kotlin 项目。

步骤一 build.gradle 基本配置。

IDEA 自动生成的 Gradle 配置文件如下：

```
group 'com.easy.kotlin'
version '1.0-SNAPSHOT'

buildscript {
    ext.kotlin_version = '1.1.2-2'

    repositories {
        mavenCentral()
    }
    dependencies {
        // Kotlin Gradle 插件
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"

    }
```

```
}

apply plugin: 'java'
apply plugin: 'kotlin'

sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre8:$kotlin_version"
    testCompile group: 'junit', name: 'junit', version: '4.12'

}
```

从上面的配置文件中可以看出，IDEA 已经自动把 Gradle 构建 Kotlin 项目插件 kotlin-gradle-plugin，以及 Kotlin 标准库 kotlin-stdlib 添加到配置文件中了。

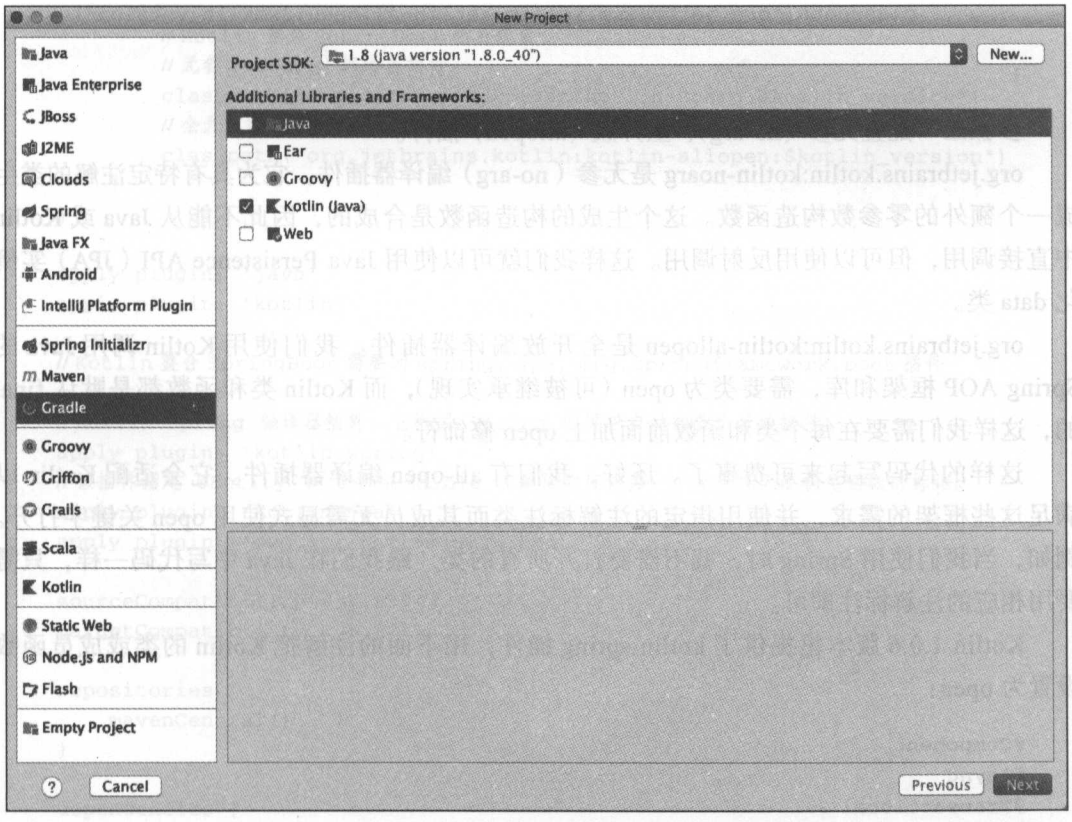


图 2-2　新建 Gradle 的 Kotlin 项目

步骤二 配置 SpringBoot 相关内容。

下面我们来配置 SpringBoot 相关内容。首先在构建脚本里面添加 ext 变量 springBoot-Version：

```
ext.kotlin_version = '1.1.2-2'
ext.springboot_version = '1.5.2.RELEASE'
```

然后在构建依赖里添加 spring-boot-gradle-plugin：

```
buildscript {
...
    dependencies {
        // Kotlin Gradle 插件
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
        // SpringBoot Gradle 插件
        classpath("org.springframework.boot:spring-boot-gradle-plugin:$springboot_
            version")

        // Kotlin 整合 SpringBoot 的默认无参构造函数，默认把所有的类设置 open 类插件
        classpath("org.jetbrains.kotlin:kotlin-noarg:$kotlin_version")
        classpath("org.jetbrains.kotlin:kotlin-allopen:$kotlin_version")
    }
}
```

步骤三 配置无参（no-arg）、全开放（allopen）插件。

org.jetbrains.kotlin:kotlin-noarg 是无参（no-arg）编译器插件，它为具有特定注解的类生成一个额外的零参数构造函数。这个生成的构造函数是合成的，因此不能从 Java 或 Kotlin 中直接调用，但可以使用反射调用。这样我们就可以使用 Java Persistence API（JPA）实例化 data 类。

org.jetbrains.kotlin:kotlin-allopen 是全开放编译器插件。我们使用 Kotlin 调用 Java 的 Spring AOP 框架和库，需要类为 open（可被继承实现），而 Kotlin 类和函数都是默认 final 的，这样我们需要在每个类和函数前面加上 open 修饰符。

这样的代码写起来可费事了。还好，我们有 all-open 编译器插件。它会适配 Kotlin 以满足这些框架的需求，并使用指定的注解标注类而其成员无需显式使用 open 关键字打开。例如，当我们使用 Spring 时，就不需要打开所有的类，跟我们在 Java 中写代码一样，只需要用相应的注解标注即可。

Kotlin 1.0.6 版本里提供了 kotlin-spring 插件，用下面的注解把 Kotlin 的类或成员函数设置为 open：

```
@Component
@Async
@Transactional
@Cacheable
```

这样的注解支持，意味着我们使用 @Configuration、@Controller、@RestController、@Service 或 @Repository 注解的类会自动设置为 open，因为这些注解都使用了 @Component。

完整的 build.gradle 配置文件如下：

```
group 'com.easy.kotlin'
version '1.0-SNAPSHOT'

buildscript {
    ext.kotlin_version = '1.1.2-2'
    ext.springboot_version = '1.5.2.RELEASE'

    repositories {
        mavenCentral()
    }
    dependencies {
        // Kotlin Gradle 插件
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
        // SpringBoot Gradle 插件
        classpath("org.springframework.boot:spring-boot-gradle-plugin:$springboot_
            version")

        // Kotlin 整合 SpringBoot 插件配置
        // 无参 (no-arg) 编译器插件
        classpath("org.jetbrains.kotlin:kotlin-noarg:$kotlin_version")
        // 全开放插件 (kotlin-allopen)
        classpath("org.jetbrains.kotlin:kotlin-allopen:$kotlin_version")
    }
}

apply plugin: 'java'
apply plugin: 'kotlin'

// Kotlin 整合 SpringBoot 需要的 spring, jpa, org.springframework.boot 插件

// Kotlin-spring 编译器插件，它根据 Spring 的要求自动配置全开放插件。
apply plugin: 'kotlin-spring'
// 该插件指定 @Entity 和 @Embeddable 注解作为应该为一个类生成无参构造函数的标记。
apply plugin: 'kotlin-jpa'
apply plugin: 'org.springframework.boot'

sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre8:$kotlin_version"
```

```
    testCompile group: 'junit', name: 'junit', version: '4.12'

    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile("org.springframework.boot:spring-boot-starter-test")
    compile("org.springframework.boot:spring-boot-starter-data-jpa")
    compile('mysql:mysql-connector-java:5.1.13')

}
```

其中，无参（no-arg）编译器插件为具有特定注解的类生成一个额外的零参数构造函数。这个生成的构造函数是合成的，因此不能从 Java 或 Kotlin 中直接调用，但可以使用反射调用。这允许 Java Persistence API（JPA）实例化 data 类，虽然它从 Kotlin 或 Java 的角度看没有无参构造函数。all-open 插件会默认把所有的类设置为 open 类。

配置 application.properties

```
spring.datasource.url = jdbc:mysql://localhost:3306/easykotlin
spring.datasource.username = root
spring.datasource.password = root
#spring.datasource.driverClassName = com.mysql.jdbc.Driver
# Specify the DBMS
spring.jpa.database = MYSQL
# Keep the connection alive if idle for a long time (needed in production)
spring.datasource.testWhileIdle = true
spring.datasource.validationQuery = SELECT 1
# Show or not log for each sql query
spring.jpa.show-sql = true
# Hibernate ddl auto (create, create-drop, update)
spring.jpa.hibernate.ddl-auto = update
# Naming strategy
spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

server.port=8000
```

2. 整体项目架构

UNIX 操作系统说："一切都是文件"。所以，我们的所有的源代码、字节码、项目资源文件等，都是文件。文件里面存的是字符串（01 也当作字符）。各种框架、库、编译器、解释器，都是对这些字符串流进行过滤，最后映射成 01 机器码（或者 CPU 微指令码等），最终落地到硬件上的高低电平。

整体项目目录如下：

```
.
├── README.md
├── build
│   └── kotlin-build
│       └── caches
```

```
│           └── version.txt
├── build.gradle
├── easykotlin.sql
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               ├── Application.kt
    │   │               ├── controller
    │   │               │   ├── HelloWorldController.kt
    │   │               │   └── PeopleController.kt
    │   │               ├── entity
    │   │               │   └── People.kt
    │   │               ├── repository
    │   │               │   └── PeopleRepository.kt
    │   │               └── service
    │   │                   └── PeopleService.kt
    │   └── resources
    │       ├── application.properties
    │       └── banner.txt
    └── test
        ├── java
        ├── kotlin
        └── resources

19 directories, 13 files
```

一切尽在不言中，静静地看项目文件结构。直接写个 HelloWorldController：

```
package com.easy.kotlin.controller

import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RestController

/**
 * Created by jack on 2017/6/7.
 */
@RestController
class HelloWorldController {
    @GetMapping(value = *arrayOf("/helloworld", "/"))
    fun helloworld(): Any {
        return "Hello,World!"
    }
}
```

我们再写个访问数据库的标准四层代码：

1）写领域模型类 People：

```
package com.easy.kotlin.entity

import java.util.*
import javax.persistence.Entity
import javax.persistence.GeneratedValue
import javax.persistence.GenerationType
import javax.persistence.Id

/**
 * Created by jack on 2017/6/6.
 */
@Entity
class People(
        @Id @GeneratedValue(strategy = GenerationType.AUTO)
        val id: Long?,
        val firstName: String?,
        val lastName: String?,
        val gender: String?,
        val age: Int?,
        val gmtCreated: Date,
        val gmtModified: Date
) {
    override fun toString(): String {
        return "People(id=$id, firstName='$firstName', lastName='$lastName',
            gender='$gender', age=$age, gmtCreated=$gmtCreated, gmtModified=
            $gmtModified)"
    }
}
```

2）写 PeopleRepository：

```
package com.easy.kotlin.repository

import com.easy.kotlin.entity.People
import org.springframework.data.repository.CrudRepository

/**
 * Created by jack on 2017/6/7.
 */
interface PeopleRepository : CrudRepository<People, Long> {
    fun findByLastName(lastName: String): List<People>?
}
```

3）写 PeopleService：

```
package com.easy.kotlin.service

import com.easy.kotlin.entity.People
import com.easy.kotlin.repository.PeopleRepository
```

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Service

/**
 * Created by jack on 2017/6/7.
 */
@Service
class PeopleService : PeopleRepository {

    @Autowired
    val peopleRepository: PeopleRepository? = null


    override fun findByLastName(lastName: String): List<People>? {
        return peopleRepository?.findByLastName(lastName)
    }

    override fun <S : People?> save(entity: S): S? {
        return peopleRepository?.save(entity)
    }

    override fun <S : People?> save(entities: MutableIterable<S>?): MutableIterable<S>? {
        return peopleRepository?.save(entities)
    }

    override fun delete(entities: MutableIterable<People>?) {
    }

    override fun delete(entity: People?) {
    }

    override fun delete(id: Long?) {
    }

    override fun findAll(ids: MutableIterable<Long>?): MutableIterable<People>? {
        return peopleRepository?.findAll(ids)
    }

    override fun findAll(): MutableIterable<People>? {
        return peopleRepository?.findAll()
    }

    override fun exists(id: Long?): Boolean {
        return peopleRepository?.exists(id)!!
    }

    override fun count(): Long {
        return peopleRepository?.count()!!
    }

    override fun findOne(id: Long?): People? {
```

```
        return peopleRepository?.findOne(id)
    }

    override fun deleteAll() {
    }

}
```

4）写 PeopleController：

```
package com.easy.kotlin.controller

import com.easy.kotlin.service.PeopleService
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.ResponseBody

/**
 * Created by jack on 2017/6/7.
 */
@Controller
class PeopleController {
    @Autowired
    val peopleService: PeopleService? = null

    @GetMapping(value = "/hello")
    @ResponseBody
    fun hello(@RequestParam(value = "lastName")
        lastName: String): Any {
        val peoples = peopleService?.findByLastName
            (lastName)
        val map = HashMap<Any, Any>()
        map.put("hello", peoples!!)
        return map
    }

}
```

3. 运行测试

首先在本地 MySQL 数据库中新建 schema：

```
CREATE SCHEMA `easykotlin` DEFAULT CHARACTER SET utf8;
```

然后，点击 Gradle 的 bootRun，如图 2-3 所示。

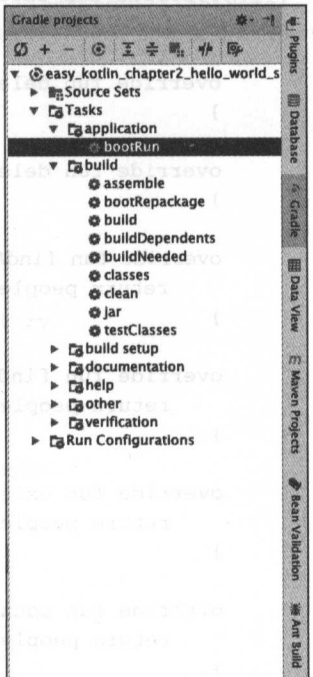


图 2-3 点击 Gradle 中的 bootRun

如果没有异常，启动成功，我们将看到的输出如图 2-4 所示。

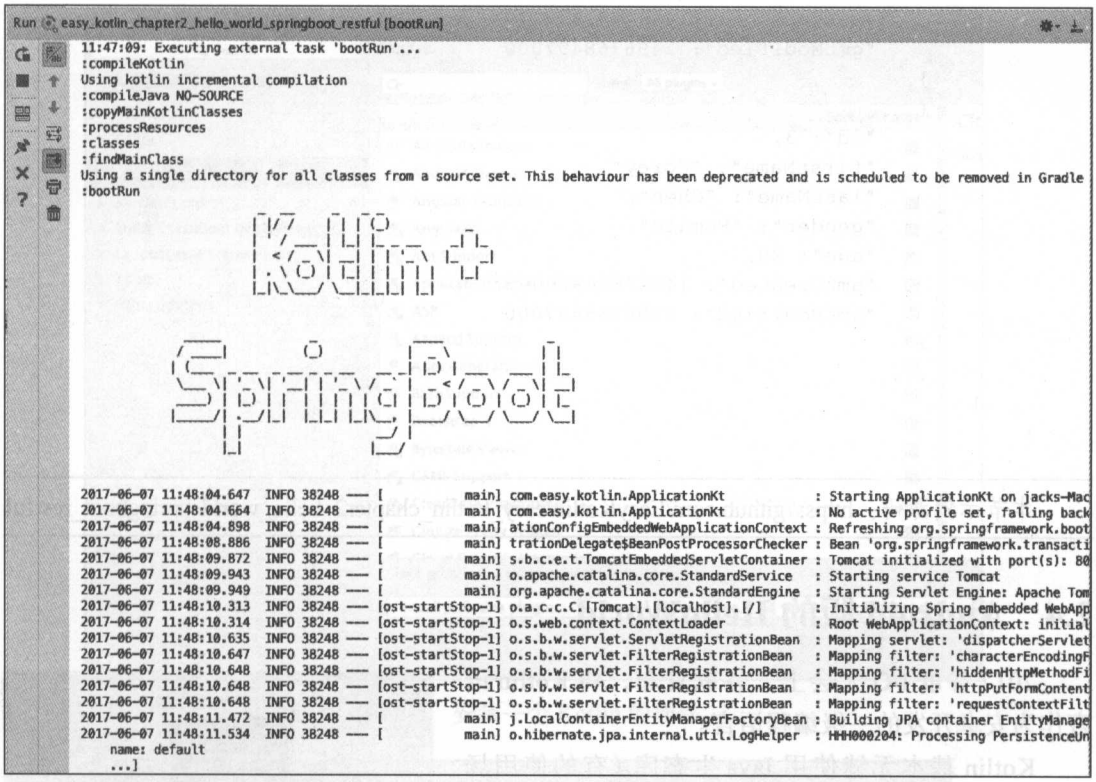


图 2-4　输出结果

打开浏览器，访问请求：

```
http://127.0.0.1:8000/
```

输出响应：

```
Hello,World!
```

访问：

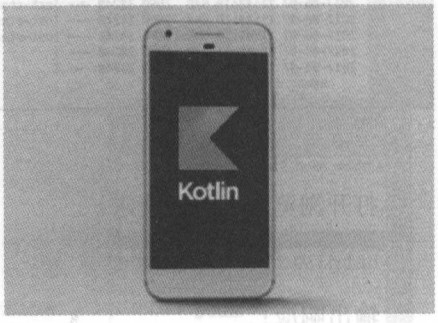
```
http://127.0.0.1:8000/hello?lastName=chen
// 20170607115700
// http://127.0.0.1:8000/hello?lastName=chen

{
    "hello": [
        {
            "id": 1,
            "firstName": "Jason",
            "lastName": "Chen",
            "gender": "Male",
            "age": 28,
```

```
            "gmtCreated": 1496768497000,
            "gmtModified": 1496768497000
        },
        {
            "id": 3,
            "firstName": "Corey",
            "lastName": "Chen",
            "gender": "Female",
            "age": 20,
            "gmtCreated": 1496768497000,
            "gmtModified": 1496768497000
        }
        ...
    ]
}
```

本节示例源码：https://github.com/EasyKotlin/easy_kotlin_chapter2_hello_world_springboot_restful

2.4 Android 版的 HelloWorld



2017 谷歌 I/O 大会上，谷歌宣布，将 Kotlin 语言作为安卓开发的一级编程语言。

Kotlin 基本无缝使用 Java 生态库（有的使用场景想要优雅地编程，可能需要插件支持，比如 no-args、allopen 插件）。使用 Kotlin 进行 Android 开发，使用 Java 生态库的过程也是类似的。

对于程序员来说，我们正处于一个美好的时代。得益于互联网的发展、工具的进步，我们现在学习一门新技术的成本和难度都比过去低了很多。

下面我们简要介绍如何在 Android 上开发一个 Kotlin 的 HelloWorld 程序。

假设你之前没有使用过 Kotlin，那么从头开始写一个 HelloWorld 的 App 也只需要如下几步。

1）要有一个 Android Studio。本节中，我们用的 Android Studio 是 2.2.3 版本，其他版本应该也大同小异：

```
Android Studio 2.3.1
Build #AI-162.3871768, built on April 1, 2017
JRE: 1.8.0_112-release-b06 x86_64
JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o
```

2）安装 Kotlin 的插件。依次打开：Android Studio → Preferences → Plugins，如图 2-5 所示。

然后选择 Browse Repositories，在搜索框中输入 Kotlin，选择结果列表中的 Kotlin 插件，如图 2-6 所示。

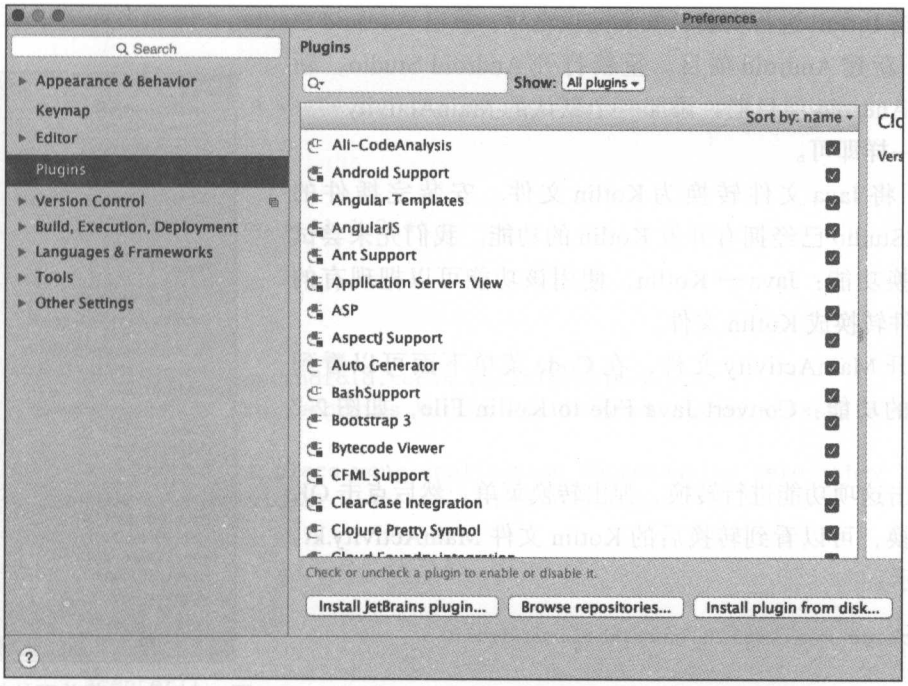


图 2-5　安装一个 Kotlin 的插件

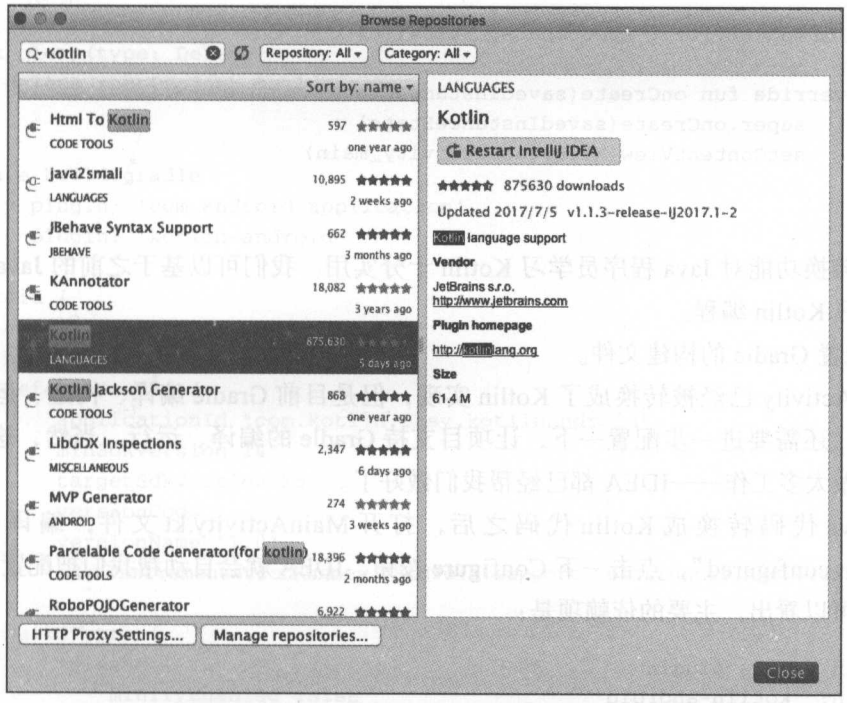


图 2-6　选择结果列表中的 Kotlin 插件，点击右侧栏中的 Install

点击 Install 进行安装，安装完成之后，重启 Android Studio。

3）新建 Android 项目。重新打开 Android Studio，新建一个 Android 项目吧，添加一个默认的 MainActivity——像以前一样即可。

4）将 Java 文件转换为 Kotlin 文件。安装完插件的 AndroidStudio 已经拥有开发 Kotlin 的功能。我们先来尝试它的转换功能：Java → Kotlin，使用该功能可以把现有的 Java 文件转换成 Kotlin 文件。

打开 MainActivity 文件，在 Code 菜单下面可以看到一个新的功能：Convert Java File to Kotlin File，如图 2-7 所示。

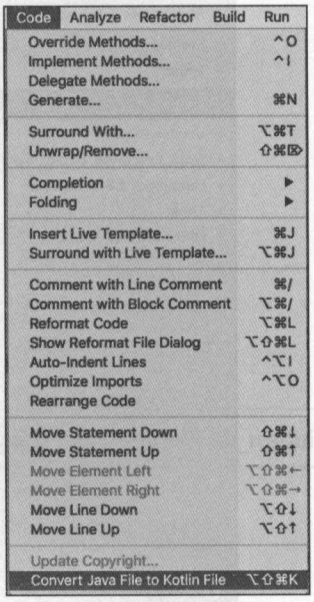


图 2-7 Code 菜单下的转换功能

点击这项功能进行转换，弹出转换菜单，然后点击 OK 进行转换，可以看到转换后的 Kotlin 文件 MainActivity.kt，如下所示：

```
package com.kotlin.easy.kotlinandroid

import android.support.v7.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

这个转换功能对 Java 程序员学习 Kotlin 十分实用。我们可以基于之前的 Java 编码经验来迅速学习 Kotlin 编程。

5）配置 Gradle 的构建文件。

MainActivity 已经被转换成了 Kotlin 实现，但是目前 Gradle 编译、构建、运行还不能执行项目，还需要进一步配置一下，让项目支持 Gradle 的编译、运行。当然，这一步也不需要我们做太多工作——IDEA 都已经帮我们做好了。

在 Java 代码转换成 Kotlin 代码之后，打开 MainActivity.kt 文件，编译器会提示“Kotlin not configured”，点击一下 Configure 按钮，IDEA 就会自动帮我们把配置文件写好。

我们可以看出，主要的依赖项是：

```
kotlin-gradle-plugin
plugin: 'kotlin-android'
kotlin-stdlib-jre7
```

完整的配置文件如下：

```
Project build.gradle
// Top-level build file where you can add configuration options common to all
   sub-projects/modules.

buildscript {
    ext.kotlin_version = '1.1.2-4'
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}

Module build.gradle
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "com.kotlin.easy.kotlinandroid"
        minSdkVersion 14
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-
```

```
                rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.3.1'
    compile 'com.android.support.constraint:constraint-layout:1.0.2'
    testCompile 'junit:junit:4.12'
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version"
}
repositories {
    mavenCentral()
}
```

可见，使用 IDEA 来写 Kotlin 代码会让你用起来如丝般润滑。毕竟 Kotlin 的亲爸爸 JetBrains 是专门做工具的，而且 Intelli IDEA 又是那么敏捷、智能。

配置之后，等 Gradle Sync 完成，即可运行。

6）运行。运行结果如图 2-8 所示。

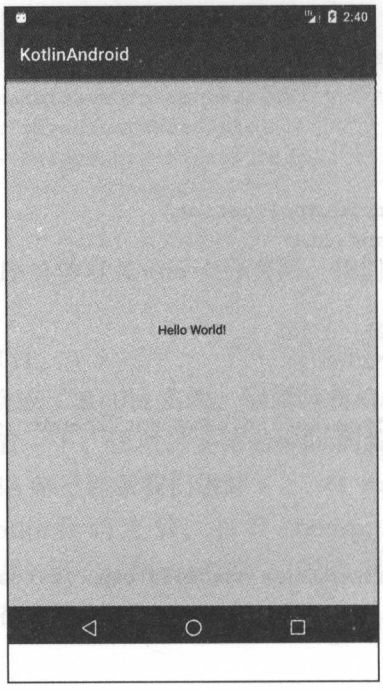


图 2-8 运行结果

本节示例源码：https://github.com/EasyKotlin/KotlinAndroid

2.5　JavaScript 版 HelloWorld

在 Kotlin 1.1 中，开始支持 JavaScript 和协程，这是引人注目的亮点。本节我们简单介绍 Kotlin 代码编译转化为 JavaScript 的方法。

为了极简直观地感受这个过程，我们先在命令行 REPL 环境体验一下 Kotlin 源码被编译生成对应的 JavaScript 代码的过程。

首先，使用编辑器新建一个 HelloWorld.kt：

```
fun helloWorld(){
    println("Hello,World!")
}
```

命令行使用 kotlinc-js 编译：

```
kotlinc-js -output HelloWorld.js HelloWorld.kt
```

运行完毕，我们会在当前目录下看到 HelloWorld.js，其内容如下：

```
if (typeof kotlin === 'undefined') {
    throw new Error("Error loading module 'HelloWorld'. Its dependency 'kotlin'
        was not found. Please, check whether 'kotlin' is loaded prior to
        'HelloWorld'.");
}
var HelloWorld = function (_, Kotlin) {
    'use strict';
    var println = Kotlin.kotlin.io.println_s8jyv4$;
    function helloWorld() {
        println('Hello,World!');
    }
    _.helloWorld = helloWorld;
    Kotlin.defineModule('HelloWorld', _);
    return _;
}(typeof HelloWorld === 'undefined' ? {} : HelloWorld, kotlin);
```

我们看到，使用 kotlinc-js 转换成的 JavaScript 代码依赖 'kotlin' 模块。这个模块是 Kotlin 支持 JavaScript 脚本的内部封装模块。也就是说，如果我们想要使用 HelloWorld.js，先要引用 kotlin.js。这个 kotlin.js 在 kotlin-stdlib-js-1.1.2.jar 里面。

下面我们使用 IDEA 新建一个 Kotlin（JavaScript）项目。在这个过程中，我们将会看到使用 Kotlin 来开发 JavaScript 应用的过程。

首先按照图 2-9 至图 2-12 所示的步骤新建项目。

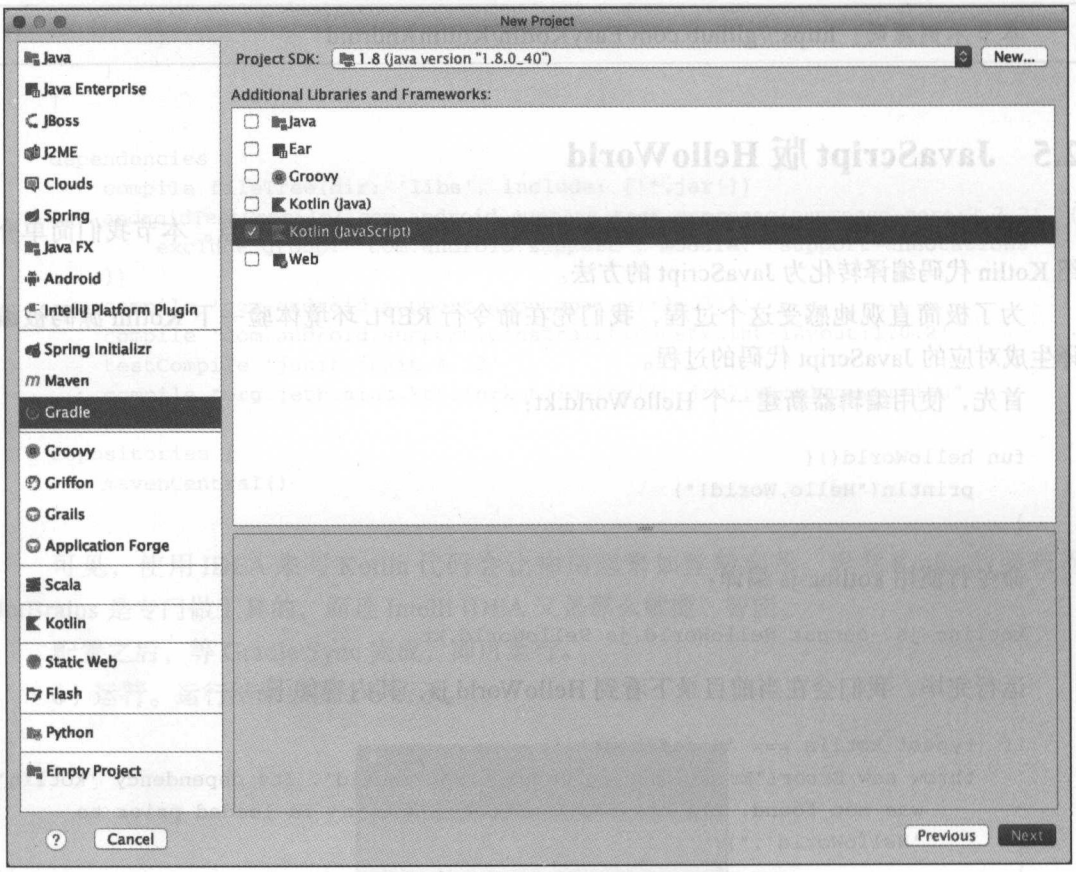


图 2-9 新建 Kotlin（JavaScript）的 Gradle 项目

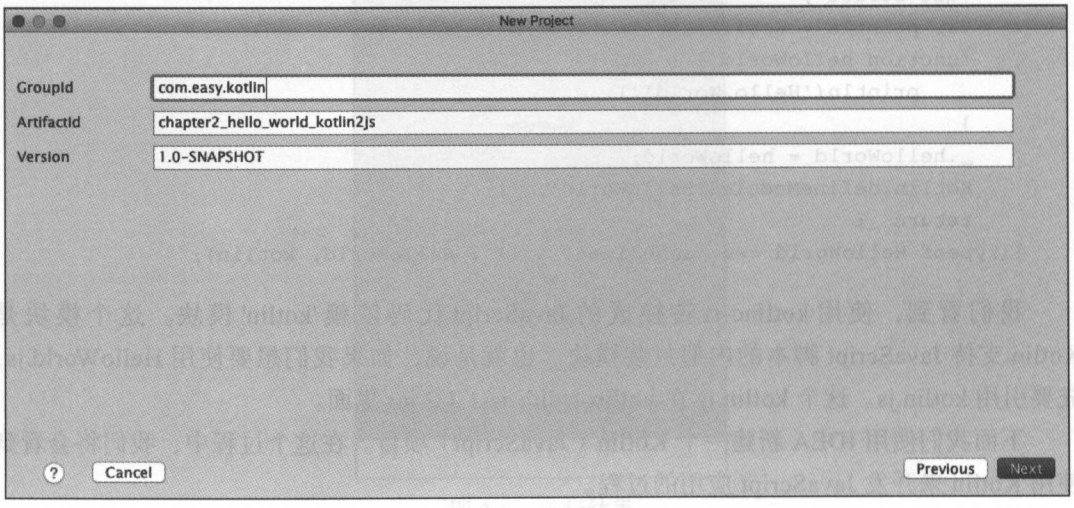


图 2-10 设置 GroupId、ArtifactId 等信息

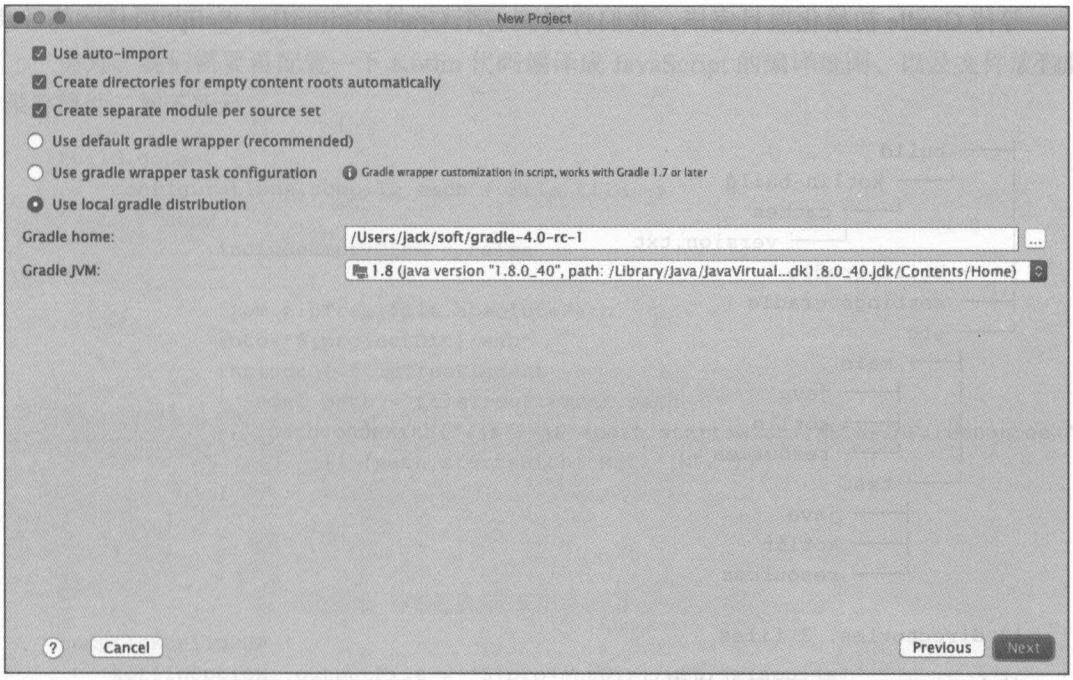


图 2-11　Gradle 环境配置

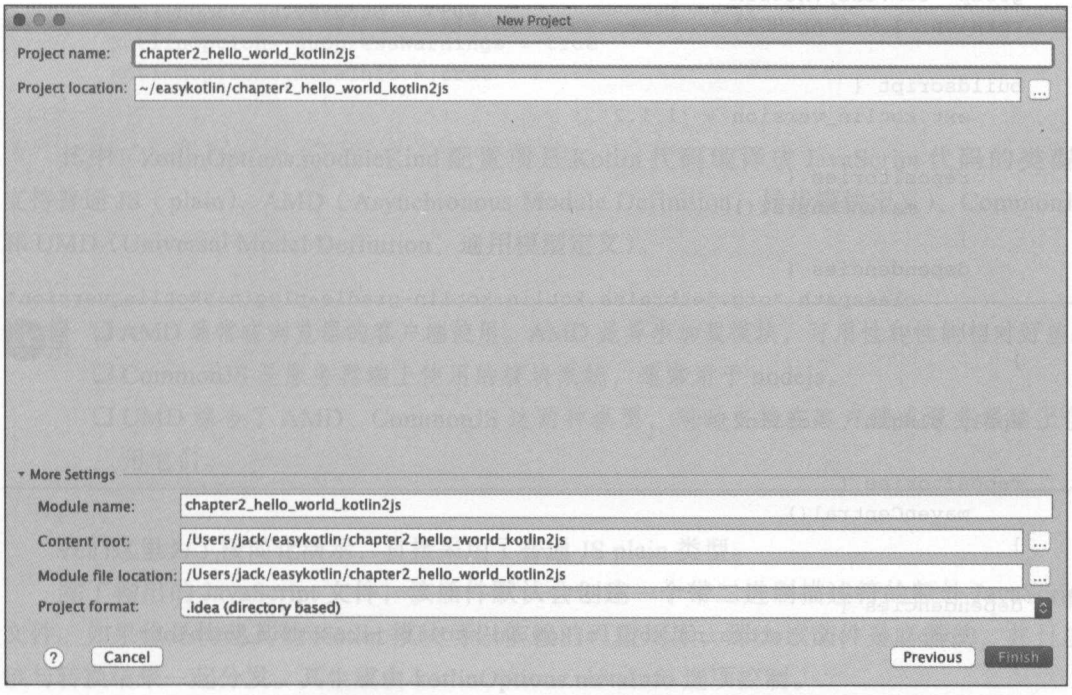


图 2-12　项目目录设置

等待 Gradle 初始化项目完毕，我们将得到一个 Gradle Kotlin JavaScript 项目，其目录如下：

```
.
├── build
│   └── kotlin-build
│       └── caches
│           └── version.txt
├── build.gradle
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   └── resources
    └── test
        ├── java
        ├── kotlin
        └── resources

12 directories, 3 files
```

其中，build.gradle 配置文件为：

```
group 'com.easy.kotlin'
version '1.0-SNAPSHOT'

buildscript {
    ext.kotlin_version = '1.1.2'

    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

apply plugin: 'kotlin2js'

repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-js:$kotlin_version"
}
```

其中，apply plugin: 'kotlin2js' 是 Gradle 的 kotlin 编译成 JavaScript 的插件。org.jetbrains.

kotlin:kotlin-stdlib-js 是 KotlinJS 的运行库。

另外，我们需要再配置一下 Kotlin 代码编译成 JavaScript 的编译规则，以及文件放置目录等属性，如下所示：

```
build.doLast {
    configurations.compile.each { File file ->
        copy {
            includeEmptyDirs = false

            from zipTree(file.absolutePath)
            into "${projectDir}/web"
            include { fileTreeElement ->
                def path = fileTreeElement.path
                path.endsWith(".js") && (path.startsWith("META-INF/resources/")
                    || !path.startsWith("META-INF/"))
            }
        }
    }
}

compileKotlin2Js {
    kotlinOptions.outputFile = "${projectDir}/web/js/app.js"
    kotlinOptions.moduleKind = "plain" //plain (default),AMD,commonjs,umd
    kotlinOptions.sourceMap = true
    kotlinOptions.verbose = true
    kotlinOptions.suppressWarnings = true
    kotlinOptions.metaInfo = true
}
```

其中，kotlinOptions.moduleKind 配置项是 Kotlin 代码编译成 JavaScript 代码的类型。支持普通 JS（plain）、AMD（Asynchronous Module Definition，异步模块定义）、CommonJS 和 UMD（Universal Model Definition，通用模型定义）。



- AMD 通常在浏览器的客户端使用。AMD 是异步加载模块，可用性和性能相对好些。
- CommonJS 是服务器端上使用的模块系统，通常用于 nodejs。
- UMD 综合了 AMD、CommonJS 这两种模型，同时支持在客户端或服务器端上使用它们。

我们这里为了极简化演示，直接采用了普通 JS plain 类型。

除了输出的 JavaScript 文件，该插件默认会创建一个带二进制描述符的额外 JavaScript 文件。如果你是构建其他 Kotlin 模块可以依赖的可重用库，那么该文件是必需的，并且应该与转换结果一起分发。其生成由 kotlinOptions.metaInfo 选项控制。

一切配置完毕，我们来写 Kotlin 代码 App.kt：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/6/7.
 */

fun helloWorld() {
    println("Hello,World!")
}
```

然后，我们直接使用 Gradle 构建项目，如图 2-13 所示。

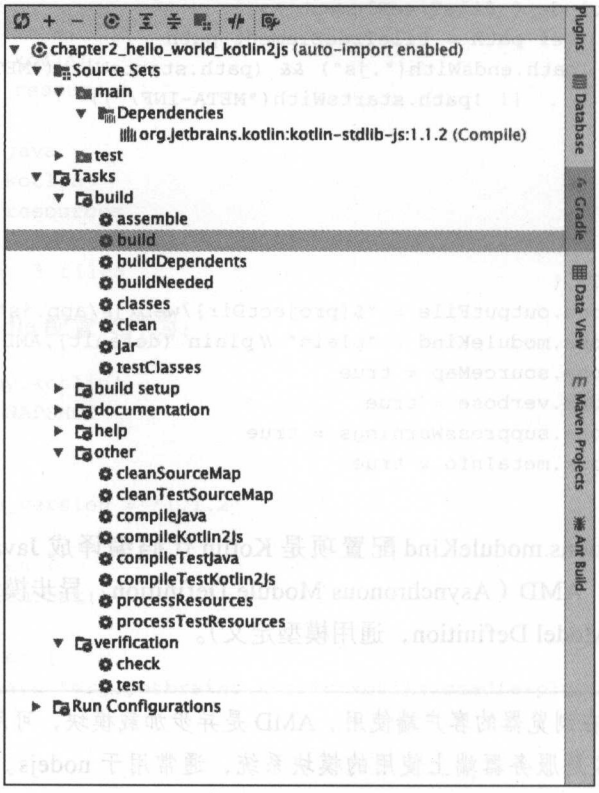


图 2-13 使用 Gradle 构建项目

控制台输出：

```
23:47:05: Executing external task 'build'...
Using a single directory for all classes from a source set. This behaviour has
    been deprecated and is scheduled to be removed in Gradle 5.0
    at build_3e0ikl0qk0r006tvk0olcp2lu.run(/Users/jack/easykotlin/chapter2_hello_
        world_kotlin2js/build.gradle:15)
:compileJava NO-SOURCE
:compileKotlin2Js
```

```
:processResources NO-SOURCE
:classes
:jar
:assemble
:compileTestJava NO-SOURCE
:compileTestKotlin2Js NO-SOURCE
:processTestResources NO-SOURCE
:testClasses UP-TO-DATE
:test NO-SOURCE
:check UP-TO-DATE
:build

BUILD SUCCESSFUL in 2s
3 actionable tasks: 3 executed
23:47:08: External task execution finished 'build'.
```

此时，我们可以看到项目目录变为：

```
.
├── build
│   └── kotlin-build
│       └── caches
│           └── version.txt
├── build.gradle
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   ├── kotlin
    │   └── resources
    └── test
        ├── java
        ├── kotlin
        └── resources

12 directories, 3 files
jack@jacks-MacBook-Air:~/easykotlin/chapter2_hello_world_kotlin2js$ tree ..
├── build
│   ├── kotlin
│   │   └── sessions
│   ├── kotlin-build
│   │   └── caches
│   │       └── version.txt
│   ├── libs
│   │   └── chapter2_hello_world_kotlin2js-1.0-SNAPSHOT.jar
│   └── tmp
│       └── jar
│           └── MANIFEST.MF
├── build.gradle
```

```
├── settings.gradle
├── src
│   ├── main
│   │   ├── java
│   │   ├── kotlin
│   │   │   └── com
│   │   │       └── easy
│   │   │           └── kotlin
│   │   │               └── App.kt
│   │   └── resources
│   └── test
│       ├── java
│       ├── kotlin
│       └── resources
└── web
    ├── js
    │   ├── app
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               └── kotlin.kjsm
    │   ├── app.js
    │   ├── app.js.map
    │   └── app.meta.js
    ├── kotlin.js
    └── kotlin.meta.js

26 directories, 14 files
```

这个 Web 目录就是 Kotlin 代码通过 kotlin-stdlib-js-1.1.2.jar 编译的输出结果。

其中，app.js 代码如下：

```
if (typeof kotlin === 'undefined') {
    throw new Error("Error loading module 'app'. Its dependency 'kotlin' was
        not found. Please, check whether 'kotlin' is loaded prior to 'app'.");
}
var app = function (_, Kotlin) {
    'use strict';
    var println = Kotlin.kotlin.io.println_s8jyv4$;
    function helloWorld() {
        println('Hello,World!');
    }
    var package$com = _.com || (_.com = {});
    var package$easy = package$com.easy || (package$com.easy = {});
    var package$kotlin = package$easy.kotlin || (package$easy.kotlin = {});
    package$kotlin.helloWorld = helloWorld;
    Kotlin.defineModule('app', _);
    return _;
```

```
}(typeof app === 'undefined' ? {} : app, kotlin);

//@ sourceMappingURL=app.js.map
```

里面这段自动生成的代码显得有点绕：

```
var package$com = _.com || (_.com = {});
    var package$easy = package$com.easy || (package$com.easy = {});
    var package$kotlin = package$easy.kotlin || (package$easy.kotlin = {});
    package$kotlin.helloWorld = helloWorld;

    Kotlin.defineModule('app', _);
    return _;
```

简化之后的意思表达如下：

```
_.com.easy.kotlin.helloWorld = helloWorld;
```

目的是建立 Kotlin 代码与 JavaScript 代码的映射关系。这样我们在前端代码中调用如下函数时：

```
function helloWorld() {
    println('Hello,World!');
}
```

只要这样调用即可：

```
app.com.easy.kotlin.helloWorld()
```

下面我们来新建一个 index.html 页面，使用我们生成的 app.js，代码如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>KotlinJS</title>

</head>
<body>


<!-- 优先加载 kotlin.js，再加载应用程序代码 app.js-->
<script type="text/javascript" src="kotlin.js"></script>
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript" src="js/app.js"></script>

<script>
    var kotlinJS = app;
    console.log(kotlinJS.com.easy.kotlin.helloWorld())
</script>
```

```
</body>
</html>
```

我们需要优先加载 kotlin.js，再加载应用程序代码 app.js。
当然，我们仍然可以像以前一样使用诸如 jquery.js 这样的库。
在浏览器中打开 index.html，如图 2-14 所示。

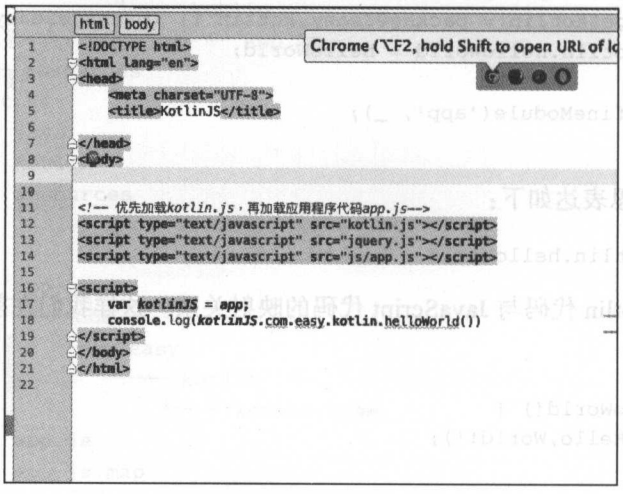


图 2-14 在浏览器中打开 index.html

我们可以看到浏览器控制台输出，如图 2-15 所示。

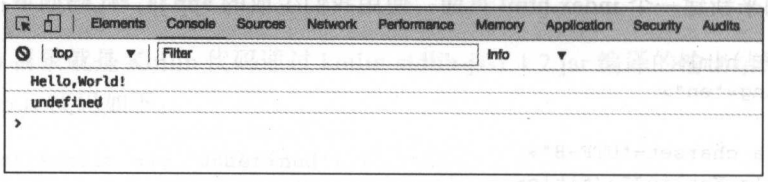


图 2-15 浏览器控制台输出

这个 helloWorld() JavaScript 函数：

```
var println = Kotlin.kotlin.io.println_s8jyv4$;
function helloWorld() {
    println('Hello,World!');
}
```

对应 kotlin.js 代码中的 3755 行处的代码：

```
BufferedOutputToConsoleLog.prototype.flush = function() {
        console.log(this.buffer);
        this.buffer = "";
    };
```

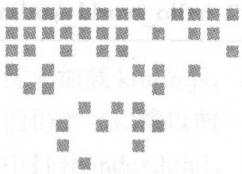
本节示例源码：https://github.com/EasyKotlin/chapter2_hello_world_kotlin2js

2.6　本章小结

本章简单介绍了 Kotlin 分别使用命令行、应用程序、Web RESTful、Android、Kotlin JavaScript 等平台上的 HelloWorld 示例。

我们体验了使用 Kotlin 代码来自动生成对应 JavaScript 代码，并在前端页面中测试使用。这非常有趣，这样使得前端 JavaScript 的代码，可以直接通过写 Kotlin 来完成了。

前两章中，我们从整体上了解了 Kotlin 语言背景，并体验了在多平台上使用 Kotlin 语言快速开发 HelloWorld 程序的过程。在下一章中，我们将学习 Kotlin 语言的基础知识。



Chapter 3 第 3 章

Kotlin 语言基础

掌握基础，持续练习。

本章介绍 Kotlin 语言的基础知识，包括关键字与标识符、表达式与流程控制、运算操作符、函数及其扩展等基本内容。我们需要通过大量实践来加深理解，以达到熟练使用。所谓“纸上得来终觉浅，绝知此事要躬行”是也。下面让我们开始吧。

3.1 包

我们先来举个例子。比如说，程序员 A 写了一个类叫 JSON，程序员 B 也写了一个类叫 JSON。然后，我们在写代码的时候，想要同时使用这两个类，该怎么区分呢？一个答案是使用目录命名空间。对应在 Java 中，就是使用包（package）来组织类，以确保类名的唯一性。上面说的例子，A 写的类放到 package com.abc.fastjson 中，B 写的类就放到 package com.bbc.jackjson 中。这样我们在代码中，就可以根据命名空间来分别使用这两个类。调用示例如下：

```
com.abc.fastjson.JSON.toJSONString()
com.bbc.jackjson.JSON.parseJSONObject()
```

Kotlin 也沿袭了 Java 的 package 概念，同时做了一些扩展。

我们可以在 *.kt 文件开头声明 package 命名空间。例如在 PackageDemo.kt 源代码中，我们按照如下方式声明包：

```
package com.easy.kotlin
```

```
/**
 * Created by jack on 2017/6/8.
 */

fun what(){
    println("This is WHAT ?")
}

class Motorbike{
    fun drive(){
        println("Drive The Motorbike ...")
    }
}

fun main(args:Array<String>){
    println("Hello,World!")
}
```

包的声明处于源文件顶部。这里，我们声明了包 com.easy.kotlin，里面定义了包级函数 what()，同时定义了一个类 Motorbike。另外，目录与包的结构无需匹配：源代码可以在文件系统的任意位置。

我们怎么使用这些类和函数呢？写一个 Junit 测试类来示例说明。首先，我们使用标准 Gradle 项目目录，对应的测试代码放在 test 目录下。具体目录结构如图 3-1 所示。

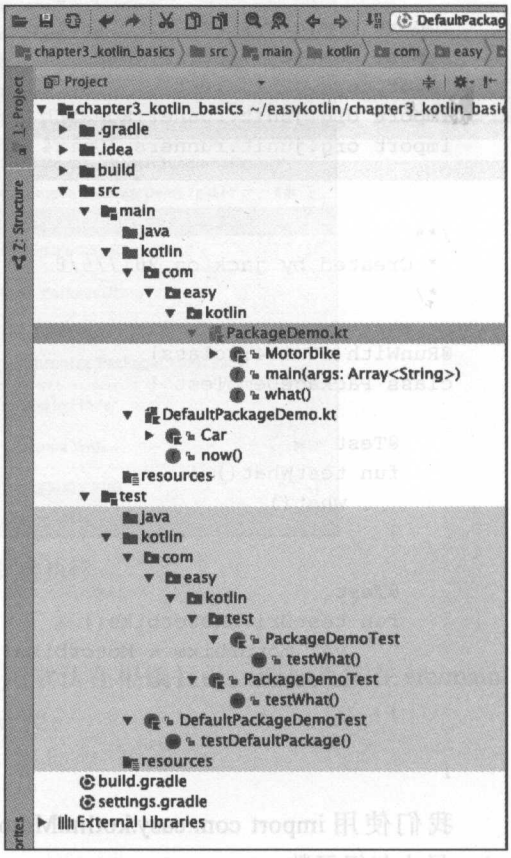


图 3-1　目录结构

我们在测试源代码目录 src/test/kotlin 下面新建一个包，跟 src/main/kotlin 在同一个 package com.easy.kotlin。然后，在此包下面新建一个测试类 PackageDemoTest：

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

/**
 * Created by jack on 2017/6/8.
 */

@RunWith(JUnit4::class)
class PackageDemoTest {

    @Test
    fun testWhat() {
```

```
        what()
    }

    @Test
    fun testDriveMotorbike(){
        val motorbike = Motorbike()
        motorbike.drive()
    }

}
```

其中，what() 函数跟 PackageDemoTest 类在同一个包命名空间下，可以直接调用，不需要 import 操作。Motorbike 类跟 PackageDemoTest 类也是同理分析。

如果不在同一个 package 下面，我们就需要 import 操作来调用对应的类和函数。例如，我们在 src/test/kotlin 目录下新建一个 package com.easy.kotlin.test，使用 package com.easy.kotlin 下面的类和函数，示例如下：

```
package com.easy.kotlin.test

import com.easy.kotlin.Motorbike
import com.easy.kotlin.what
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4


/**
 * Created by jack on 2017/6/8.
 */

@RunWith(JUnit4::class)
class PackageDemoTest {

    @Test
    fun testWhat() {
        what()
    }

    @Test
    fun testDriveMotorbike() {
        val motorbike = Motorbike()
        motorbike.drive()
    }

}
```

我们使用 import com.easy.kotlin.Motorbike 导入类，直接使用 import com.easy.kotlin.what 导入包级函数。

上面我们使用 JUnit4 测试框架。在 build.gradle 中的依赖是：

```
testCompile group: 'junit', name: 'junit', version: '4.12'
```

右击测试类，点击 Run 可以执行，如图 3-2 所示。

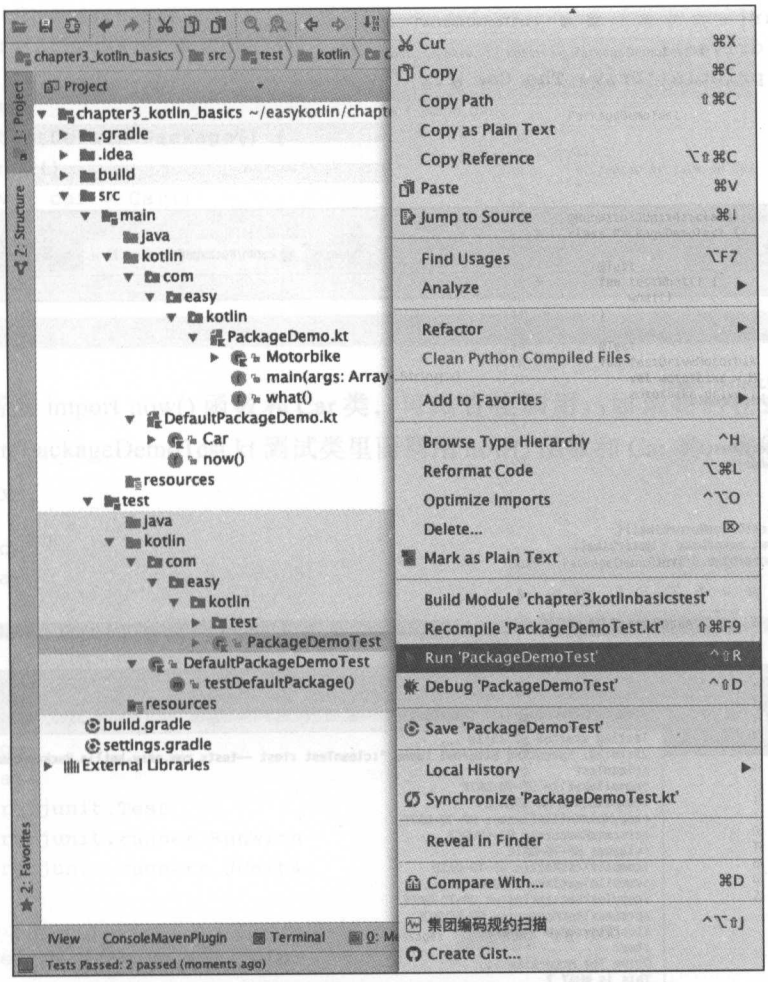


图 3-2　右击测试类执行

运行结如图 3-3 所示。

另外，如果我们不定义 package 命令空间，则默认在根级目录。例如直接在 src/main/kotlin 源代码目录下面新建 DefaultPackageDemo.kt 类：

```
import java.util.*

/**
 * Created by jack on 2017/6/8.
```

```
 */

fun now() {
    println("Now Date is: " + Date())
}

class Car{
    fun drive(){
        println("Drive The Car ... ")
    }

}
```

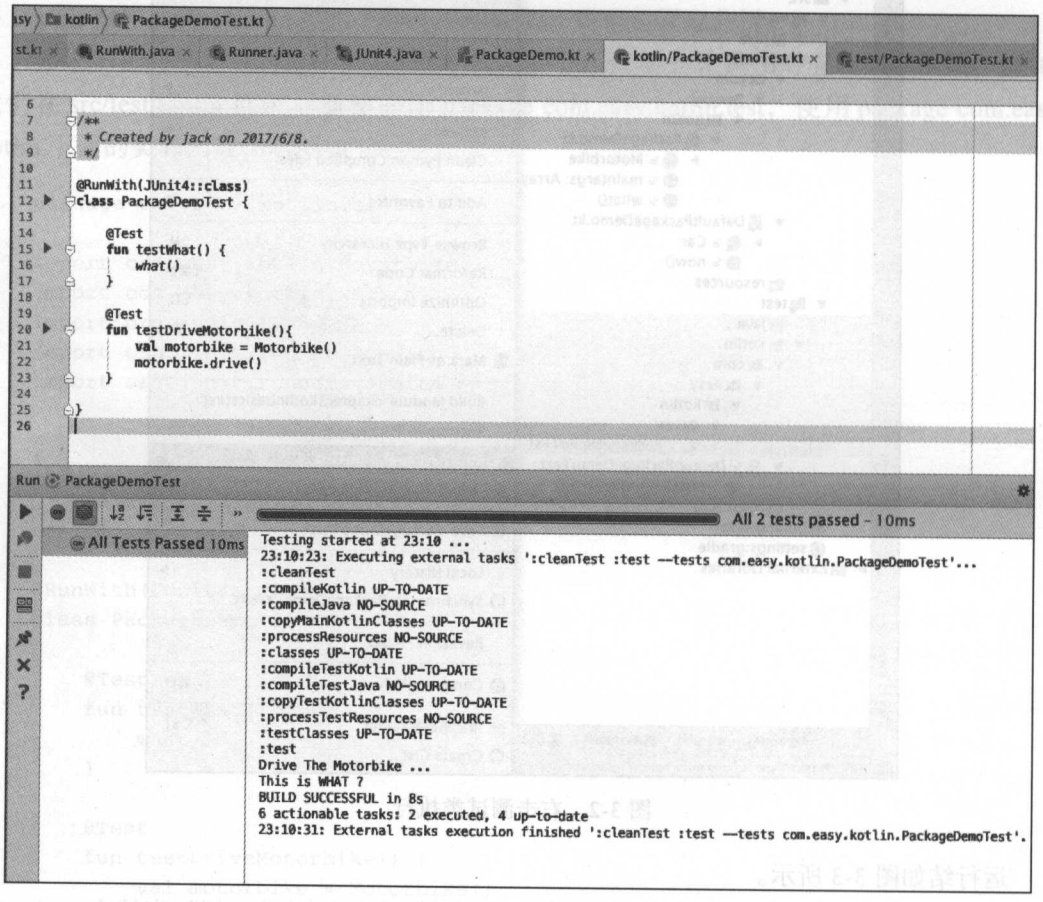


图 3-3 运行结果

如果，我们同样在 src/test/kotlin 目录下面新建测试类 DefaultPackageDemoTest：

```
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4
```

```
/**
 * Created by jack on 2017/6/8.
 */


@RunWith(JUnit4::class)
class DefaultPackageDemoTest {

    @Test
    fun testDefaultPackage() {
        now()
        val car = Car()
        car.drive()
    }


}
```

我们不需要 import now() 函数和 Car 类，可以直接调用。如果我们在 src/test/kotlin/com/easy/kotlin/PackageDemoTest.kt 测试类里面调用 now() 函数和 Car 类，我们按照下面的方式进行 import：

```
import now
import Car
```

PackageDemoTest.kt 完整测试代码如下：

```
package com.easy.kotlin

import now
import Car
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4


/**
 * Created by jack on 2017/6/8.
 */

@RunWith(JUnit4::class)
class PackageDemoTest {

    @Test
    fun testWhat() {
        what()
    }

    @Test
    fun testDriveMotorbike(){
```

```
        val motorbike = Motorbike()
        motorbike.drive()
    }

    @Test
    fun testDefaultPackage() {
        now()
        val car = Car()
        car.drive()
    }

}
```

另外，Kotlin 会针对每个 Kotlin 文件默认导入一些基础包：

- kotlin.*
- kotlin.annotation.*
- kotlin.collections.*
- kotlin.comparisons.*（自 1.1 起）
- kotlin.io.*
- kotlin.ranges.*
- kotlin.sequences.*
- kotlin.text.*

根据目标平台还会导入额外的包：

- JVM：
  java.lang.*
  kotlin.jvm.*
- JS：
  kotlin.js.*

本节示例代码：https://github.com/EasyKotlin/chapter3_kotlin_basics/tree/package_demo

3.2 声明变量和值

首先，在 Kotlin 中，一切都是对象。所以，所有变量也都是对象（也就是说，任何变量都是根据引用类型来使用的）。

Kotlin 的变量分为 var（可变的）和 val（不可变的）。可以简单理解为：

- var 是可写的，在它生命周期中可以被多次赋值；
- 而 val 是只读的，仅能一次赋值，后面就不能被重新赋值。

代码示例如下：

```
package com.easy.kotlin

import java.util.*

/**
 * Created by jack on 2017/6/8.
 */

class VariableVSValue {
    fun declareVar() {
        var a = 1
        a = 2
        println(a)
        println(a::class)
        println(a::class.java)

        var x = 5        //自动推断出 `Int` 类型
        x += 1

        println("x = $x")
    }

    fun declareVal() {
        val b = "a"
        //b  = "b"        //编译器会报错: Val cannot be reassigned
        println(b)
        println(b::class)
        println(b::class.java)

        val c: Int = 1   //立即赋值
        val d = 2        //自动推断出 `Int` 类型
        val e: Int       //如果没有初始值类型不能省略
        e = 3            //明确赋值
        println("c = $c, d = $d, e = $e")
    }
}
```

我们知道，在Java中也分可变与不可变（final）。在Kotlin中，更简洁的、更常用的场景是：只要可能，尽量在Kotlin中首选使用val不变值。因为事实上在程序中大部分地方使用不可变的变量能带来很多益处，如：可预测的行为和线程安全。

3.3 变量类型推断

3.3.1 省去变量类型

在Kotlin中，大部分情况不需要说明你使用对象的类型，编译器可以直接推断出它的类型。代码示例如下：

```
fun typeInference(){
    val str = "abc"
    println(str)
    println(str is String)
    println(str::class)
    println(str::class.java)

    //abc
    //true
    //class java.lang.String (Kotlin reflection is not available)
    //class java.lang.String

    val d = Date()
    println(d)
    println(d is Date)
    println(d::class)
    println(d::class.java)

    //Fri Jun 09 00:06:33 CST 2017
    //true
    //class java.util.Date (Kotlin reflection is not available)
    //class java.util.Date

    val bool = true
    println(bool)
    println(bool::class)
    println(bool::class.java)

    //true
    //boolean (Kotlin reflection is not available)
    //boolean

    val array = arrayOf(1,2,3)
    println(array)
    println(array is Array)
    println(array::class)
    println(array::class.java)

    //[Ljava.lang.Integer;@7b5eadd8
    //true
    //class [Ljava.lang.Integer; (Kotlin reflection is not available)
    //class [Ljava.lang.Integer;
}
```

所以，我们只需要依据要产生的变量类型填写 var 或 val，其类型通常能够被推断出来。编译器能够检测到其类型，自动地完成类型转换。当然，我们也可以明确地指定变量类型。

但是，类型推断不是所有的。例如，整型变量 Int 不能赋值 Long 变量。下面的代码不能通过编译：

```
fun Int2Long(){
    val x:Int = 10
    val y:Long = x //Type mismatch
}
```

我们需要显式地调用对应的类型转换函数进行转换：

```
fun Int2Long(){
    val x:Int = 10
    val y:Long = x //Type mismatch
    val y: Long = x.toLong()
}
```

3.3.2　使用 is 运算符进行类型检测

is 运算符检测一个表达式是否为某类型的一个实例。

如果一个不可变的局部变量或属性已经判断出为某类型，那么检测后的分支中可以直接当作该类型使用，无需显式转换：

```
fun getLength(obj: Any): Int? {
    var result = 0
    if (obj is String) {
        //`obj` 在该条件分支内自动转换成 `String`
        println(obj::class)      //class java.lang.String
        result = obj.length
        println(result)
    }
    //在离开类型检测分支后，`obj` 仍然是 `Any` 类型
    println(obj::class)          //class java.lang.Object
    return result
}
```

测试类如下：

```
@Test
fun testGetLength() {
    val obj = "abcdef"
    val len = getLength(obj)
    Assert.assertTrue(len == 6)

    val obj2:Any = Any()
    getLength(obj2)

}
```

3.4　字符串与其模板表达式

原始字符串（raw string）由三重引号（"""）分隔（这个跟 Python 一样）。原始字符串可

以包含换行符和任何其他字符：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/6/9.
 */


fun main(args: Array<String>) {
    val rawString = """
fun helloWorld(val name : String) {
    println("Hello, world!")
}
"""
    println(rawString)

}
```

字符串可以包含模板表达式。模板表达式以美元符号（$）开始：

```
val fooTemplateString = "$rawString has ${rawString.length} characters"
println(fooTemplateString)
```

3.5 流程控制语句

流程控制语句是编程语言中的核心之一，一般可分为如下三类：

- 分支语句（if、when）
- 循环语句（for、while）
- 跳转语句（break、continue、return、throw）

另外，在跳转语句中，为了更加灵活地控制程序要跳转的目的地，Kotlin 还引入了标签的概念。下面就分别来介绍一下。

3.5.1 if 表达式

if-else 语句是控制程序流程的最基本形式，其中 else 是可选的。在 Kotlin 中，if 是一个表达式，即它会返回一个值（跟 Scala 一样）。

代码示例如下：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/6/9.
 */


fun main(args: Array<String>) {
```

```
    println(max(1, 2))
}

fun max(a: Int, b: Int): Int {

//作为表达式
    val max = if (a > b) a else b
    return max // return if (a > b) a else b
}

fun max1(a: Int, b: Int): Int {
    //传统用法
    var max1 = a
    if (a < b) max1 = b
    return max1

}

fun max2(a: Int, b: Int): Int {

//With else
    var max2: Int
    if (a > b) {
        max2 = a
    } else {
        max2 = b
    }
    return max2
}
```

另外，if的分支可以是代码块，最后的表达式作为该块的值：

```
fun max3(a: Int, b: Int): Int {
    val max = if (a > b) {
        print("Max is a")
        a
    } else {
        print("Max is b")
        b
    }
    return max
}
```

if作为代码块时，最后一行为其返回值。

另外，在Kotlin中没有类似true?1:0这样的三元表达式。对应的写法是使用if else语句：

```
if(true) 1 else 0
```

如果if表达式只有一个分支，或者分支的结果是Unit，它的值就是Unit。

示例代码如下：

```
>>> val x = if(1==1) true
>>> x
kotlin.Unit
>>> val y = if(1==1) true else false
>>> y
true
```

if-else 语句规则：

1）if 后的括号不能省略，括号里表达式的值须是布尔型。

代码反例：

```
>>> if("a") 1
error: type mismatch: inferred type is String but Boolean was expected
if("a") 1
    ^

>>> if(1) println(1)
error: the integer literal does not conform to the expected type Boolean
if(1)
    ^
```

2）如果条件体内只有一条语句需要执行，那么 if 后面的大括号可以省略。良好的编程风格建议加上大括号：

```
>>> if(true) println(1) else println(0)
1
>>> if(true) { println(1)}  else{ println(0)}
1
```

3）对于给定的 if，else 语句是可选的，else if 语句也是可选的。

4）else 和 else if 同时出现时，else 必须出现在 else if 之后。

5）如果有多条 else if 语句同时出现，那么如果有一条 else if 语句的表达式测试成功，那么会忽略掉其他所有 else if 和 else 分支。

6）如果出现多个 if，只有一个 else 的情形，else 子句归属于最内层的 if 语句。

以上规则跟 Java、C 语言基本相同。

3.5.2 when 表达式

when 表达式类似于 switch-case 表达式。when 会对所有的分支进行检查直到有一个条件满足为止。但相比 switch 而言，when 语句要更加强大、灵活。

Kotlin 的极简语法表达风格，使得我们对分支检查的代码写起来更加简单直接：

```
fun cases(obj: Any) {
    when (obj) {
```

```
        1 -> print(" 第一项 ")
        "hello" -> print(" 这个是字符串 hello")
        is Long -> print(" 这是一个 Long 类型数据 ")
        !is String -> print(" 这不是 String 类型的数据 ")
        else -> print("else 类似于 Java 中的 default")
    }
}
```

像 if 一样，when 的每一个分支也可以是一个代码块，它的值是块中最后的表达式的值。

如果其他分支都不满足条件会到 else 分支（类似 default）。如果我们有很多分支需要用相同的方式处理，则可以把多个分支条件放在一起，用逗号分隔：

```
fun switch(x: Any) {
    when (x) {
        -1, 0 -> print("x == -1 or x == 0")
        1 -> print("x == 1")
        2 -> print("x == 2")
        else -> { // 注意这个块
            print("x is neither 1 nor 2")
        }
    }
}
```

可以用任意表达式（而不只是常量）作为分支条件：

```
fun switch(x: Int) {
    val s = "123"
    when (x) {
        -1, 0 -> print("x == -1 or x == 0")
        1 -> print("x == 1")
        2 -> print("x == 2")
        8 -> print("x is 8")
        parseInt(s) -> println("x is 123")
        else -> { // 注意这个块
            print("x is neither 1 nor 2")
        }
    }
}
```

也可以检测一个值在（in）或者不在（!in）一个区间或者集合中：

```
val x = 1
val validNumbers = arrayOf(1, 2, 3)
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

3.5.3 for 循环

Kotlin 的 for 循环与现代的程序设计语言基本相同。

for 循环可以对任何提供迭代器（iterator）的对象进行遍历，语法如下：

```
for (item in collection) {
    print(item)
}
```

循环体可以是一个代码块，例如：

```
for (i in intArray) {
    ...
}
```

代码示例如下：

```
/**
 * For loop iterates through anything that provides an iterator.
 * See http://kotlinlang.org/docs/reference/control-flow.html#for-loops
 */
fun main(args: Array<String>) {
    for (arg in args)
        println(arg)
    // or
    println()
    for (i in args.indices)
        println(args[i])
}
```

如果想要通过索引遍历一个数组或者一个 list，可以这么做：

```
for (i in array.indices) {
    print(array[i])
}
```

或者可以用库函数 withIndex 操作：

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

3.5.4 while 循环

while 和 do .. while 使用方式与 C、Java 语言基本一致，代码示例如下：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/6/9.
 */
```

```
fun main(args: Array<String>) {
    var x = 10
    while (x > 0) {
        x--
        println(x)
    }

    var y = 10
    do {
        y = y + 1
        println(y)
    } while (y < 20) //y 的作用域包含此处
}
```

3.5.5　break 和 continue

break 和 continue 都是用来控制循环结构的，主要是用来停止循环（中断跳转）。

1. break

我们在写代码的时候，经常会遇到在某种条件出现时，就直接提前终止循环。而不是等到循环条件为 false 时才终止。这个时候，我们就可以使用 break 结束循环。break 用于完全结束一个循环，直接跳出循环体，然后执行循环体后面的语句。

问题场景：打印数字 110，只要遇到偶数，就结束打印。

代码示例如下：

```
fun breakDemo_1() {
    for (i in 1..10) {
        println(i)
        if (i % 2 == 0) {
            break
        }
    } //break to here
}
```

测试代码：

```
@Test
fun testBreakDemo_1(){
    breakDemo_1()
}
```

输出：

```
1
2
```

2. continue

continue 是只终止本轮循环，但是还会继续下一轮循环。可以简单理解为，直接在当前

语句处中断，跳转到循环入口，执行下一轮循环；而 break 则是跳转到循环出口。

问题场景：打印数字 010，但是不打印偶数。

代码示例如下：

```
fun continueDemo() {
    for (i in 1..10) {
        if (i % 2 == 0) {
            continue
        }
        println(i)
    }
}
```

测试代码：

```
@Test
fun testContinueDemo() {
    continueDemo()
}
```

输出：

```
1
3
5
7
9
```

3.5.6 return

在 Java、C 语言中，return 语句是我们再常见不过的了。虽然在 Scala、Groovy 这样的语言中，函数的返回值可以不需要显示地用 return 来指定，但是我们仍然认为，使用 return 的编码风格更加容易阅读理解。

在 Kotlin 中，除了表达式的值，有返回值的函数都要求显式地使用 return 来返回其值。

代码示例如下：

```
fun sum(a: Int,b: Int): Int{
    return a+b
}

fun max(a: Int, b: Int): Int { if (a > b) return a else return b}
```

在 Kotlin 中，可以直接使用 = 符号来直接返回一个函数的值。

代码示例如下：

```
>>> fun sum(a: Int,b: Int) = a + b
>>> fun max(a: Int, b: Int) = if (a > b) a else b

>>> sum(1,10)
```

```
11

>>> max(1,2)
2

>>> val sum=fun(a:Int, b:Int) = a+b
>>> sum
(kotlin.Int, kotlin.Int) -> kotlin.Int
>>> sum(1,1)
2

>>> val sumf = fun(a:Int, b:Int) = {a+b}
>>> sumf
(kotlin.Int, kotlin.Int) -> () -> kotlin.Int
>>> sumf(1,1)
() -> kotlin.Int
>>> sumf(1,1).invoke()
2
```

上述代码示例中，我们可以看到，后面的函数体语句有没有大括号 {} 意思完全不同。加了大括号，意义就完全不一样了，通过下面的代码示例可清晰地看出：

```
>>> fun sumf(a:Int,b:Int) = {a+b}
>>> sumf(1,1)
() -> kotlin.Int
>>> sumf(1,1).invoke
error: function invocation 'invoke()' expected
sumf(1,1).invoke
          ^
>>> sumf(1,1).invoke()
2
>>> fun maxf(a:Int, b:Int) = {if(a>b) a else b}
>>> maxf(1,2)
() -> kotlin.Int
>>> maxf(1,2).invoke()
2
```

可以看出，sumf、maxf 的返回值是函数类型：

```
() -> kotlin.Int
() -> kotlin.Int
```

这点跟 Scala 是不同的。在 Scala 中，带不带大括号 {}，意思一样：

```
scala> def maxf(x:Int, y:Int) = { if(x>y) x else y }
maxf: (x: Int, y: Int)Int

scala> def maxv(x:Int, y:Int) = if(x>y) x else y
maxv: (x: Int, y: Int)Int

scala> maxf(1,2)
```

```
res4: Int = 2

scala> maxv(1,2)
res6: Int = 2
```

我们可以看出 maxf: (x: Int, y: Int)Int 跟 maxv: (x: Int, y: Int)Int 签名是一样的。在这里，Kotlin 跟 Scala 在大括号的使用上，是完全不同的。

然后，调用方式是直接调用 invoke() 函数。通过 REPL 的编译错误提示信息，我们也可以看出，在 Kotlin 中，调用无参函数也是要加上括号 () 的。

Kotlin 中 return 语句会从最近的函数或匿名函数中返回，但是在 Lambda 表达式中遇到 return，则直接返回最近的外层函数。例如下面两个函数是不同的：

```
fun returnDemo_1() {
    println(" START " + ::returnDemo_1.name)
    val intArray = intArrayOf(1, 2, 3, 4, 5)
    intArray.forEach {
        if (it == 3) return
        println(it)
    }
    println(" END " + ::returnDemo_2.name)
}

//1
//2

fun returnDemo_2() {
    println(" START " + ::returnDemo_2.name)
    val intArray = intArrayOf(1, 2, 3, 4, 5)
    intArray.forEach(fun(a: Int) {
        if (a == 3) return
        println(a)
    })
    println(" END " + ::returnDemo_2.name)
}

//1
//2
//4
//5
```

returnDemo_1 在遇到 3 时会直接返回（有点类似循环体中的 break 行为）。最后输出为：

```
1
2
```

returnDemo_2 遇到 3 时会跳过它继续执行（有点类似循环体中的 continue 行为）。最后输出为：

```
1
```

```
2
4
5
```

在 returnDemo_2 中，我们用一个匿名函数替代 lambda 表达式。匿名函数内部的 return 语句将从该匿名函数自身返回。

在 Kotlin 中，这是匿名函数和 lambda 表达式行为不一致的地方。当然，为了显式地指明 return 返回的地址，为此 Kotlin 还提供了 @Label（标签）来控制返回语句，且看下节分解。

3.5.7　标签

在 Kotlin 中任何表达式都可以用标签（label）来标记。标签的格式为标识符后跟 @ 符号，例如：abc@、jarOfLove@ 都是有效的标签。我们可以用 Label 标签来控制 return、break 或 continue 的跳转（jump）行为。

Kotlin 的函数是可以被嵌套的。它有函数字面量、局部函数等。有了标签限制的 return，我们就可以从外层函数返回了。例如，从 lambda 表达式中返回，returnDemo_2() 我们可以显示地指定 lambda 表达式中的 return 地址是其入口处。

代码示例如下：

```
fun returnDemo_3() {
    println(" START " + ::returnDemo_3.name)
    val intArray = intArrayOf(1, 2, 3, 4, 5)
    intArray.forEach here@ {
        if (it == 3) return@here          // 指令跳转到 lambda 表达式标签 here@ 处。
                                          // 继续下一个 it=4 的遍历循环
        println(it)
    }
    println(" END " + ::returnDemo_3.name)
}

//1
//2
//4
//5
```

我们在 lambda 表达式开头处添加了标签 here@，可以这么理解：该标签相当于是记录了 Lambda 表达式的指令执行入口地址，然后在表达式内部使用 return@here 跳转至 Lambda 表达式该地址处。

另外，也可以使用隐式标签更方便，该标签与接收该 lambda 的函数同名。

代码示例如下：

```
fun returnDemo_4() {
    println(" START " + ::returnDemo_4.name)
```

```
    val intArray = intArrayOf(1, 2, 3, 4, 5)
    intArray.forEach {
        if (it == 3) return@forEach // 从 lambda 表达式 @forEach 中返回。
        println(it)
    }
    println(" END " + ::returnDemo_4.name)
}
```

接收该 Lambda 表达式的函数是 forEach，所以可以直接使用 return@forEach 跳转到此处执行下一轮循环。

通常当我们在循环体中使用 break，是跳出最近外层的循环，如下所示：

```
fun breakDemo_1() {
    println("---------------  breakDemo_1 ---------------")
    for (outer in 1..5) {
        println("outer=" + outer)
        for (inner in 1..10) {
            println("inner=" + inner)
            if (inner % 2 == 0) {
                break
            }
        }
    }
}
```

输出为：

```
                              -  breakDemo_1 ---------------
outer=1
inner=1
inner=2
outer=2
inner=1
inner=2
outer=3
inner=1
inner=2
outer=4
inner=1
inner=2
outer=5
inner=1
inner=2
```

当我们想直接跳转到外层 for 循环，这个时候我们就可以使用标签了。

代码示例如下：

```
fun breakDemo_2() {
    println("---------------  breakDemo_2 ---------------")
```

```
    outer@ for (outer in 1..5)
        for (inner in 1..10) {
            println("inner=" + inner)
            println("outer=" + outer)
            if (inner % 2 == 0) {
                break@outer
            }
        }
}
```

输出为：

```
---------- breakDemo_2 ----------
inner=1
outer=1
inner=2
outer=1
```

有时候，为了代码可读性，可以用标签来显式地指出循环体的跳转地址，比如在breakDemo_1()中，我们可以用标签来指明内层循环的跳转地址：

```
fun breakDemo_3() {
    println("---------- breakDemo_3 ----------")
    for (outer in 1..5)
        inner@ for (inner in 1..10) {
            println("inner=" + inner)
            println("outer=" + outer)
            if (inner % 2 == 0) {
                break@inner
            }
        }
}
```

3.5.8 throw表达式

在Kotlin中throw是表达式，它的类型是特殊类型：Nothing，该类型没有值。与C、Java中的void意思一样：

```
>>> Nothing::class
class java.lang.Void
```

我们在代码中，用Nothing来标记无返回的函数：

```
>>> fun fail(msg:String):Nothing{ throw IllegalArgumentException(msg) }
>>> fail("XXXX")
java.lang.IllegalArgumentException: XXXX
    at Line57.fail(Unknown Source)
```

另外，如果把一个throw表达式的值赋值给一个变量，需要显式声明类型为Nothing，代码示例如下：

```
>>> val ex = throw Exception("YYYYYYYY")
error: 'Nothing' property type needs to be specified explicitly
val ex = throw Exception("YYYYYYYY")
    ^

>>> val ex:Nothing = throw Exception("YYYYYYYY")
java.lang.Exception: YYYYYYYY
```

另外，因为 ex 变量是 Nothing 类型，没有任何值，所以无法当做参数传给函数：

```
>>> println(ex)
error: overload resolution ambiguity:
@InlineOnly public inline fun println(message: Any?): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Boolean): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Byte): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Char): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: CharArray): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Double): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Float): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Int): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Long): Unit defined in kotlin.io
@InlineOnly public inline fun println(message: Short): Unit defined in kotlin.io
println(ex)
^

>>> ex
exception: org.jetbrains.kotlin.codegen.CompilationException: Back-end (JVM)
    Internal error: Unregistered script: class Line62
Cause: Unregistered script: class Line62
File being compiled and position: (1,1) in /line64.kts
PsiElement: ex
The root cause was thrown at: ScriptContext.java:86
...
```

3.6 代码注释

正如 Java 和 JavaScript，Kotlin 支持行注释及块注释：

```
// 这是一个行注释

/* 这是一个多行的
   块注释。*/
```

与 Java 不同的是，Kotlin 的块注释可以嵌套。就是说，你可以这样注释：

```
/**
 * hhhh
 * /**
 *  fff
 *  /**
```

```
 *     ggggg
 *  */
 * */
 *
 * abc
 *
 */
fun main(args:Array<String>){
    val f = Functions()
    println(f.fvoid1())
    println(f.fvoid2())
    println(f.sum1(1,1))
    println(f.sum2(1,1))
}
```

3.7　语法与标识符

我们知道，任何一门编程语言都会有一些自己专用的关键字、符号以及规定的语法规则等等。程序员们使用这些基础词汇和语法规则来表达算法步骤，也就是写代码的过程。

词法分析是编译器对源码进行编译的基础步骤之一。词法分析是将源程序读入的字符序列，按照一定的规则转换成词法单元（Token）序列的过程。词法单元是语言中具有独立意义的最小单元，包括修饰符、关键字、常数、操作符，等等。

3.7.1　修饰符

在 Kotlin 源码项目中的 kotlin/grammar/src/modifiers.grm 文件中，描述了 Kotlin 语言的修饰符，我们在此作简要注释说明：

```
/**
## Modifiers
*/

modifiers
    : (modifier | annotations)*
    ;

typeModifiers
    : (suspendModifier | annotations)*
    ;

modifier
    : classModifier
    : accessModifier
    : varianceAnnotation
    : memberModifier
    : parameterModifier
    : typeParameterModifier
```

```
    : functionModifier
    : propertyModifier
    ;

classModifier 类修饰符
    : "abstract" 抽象类
    : "final" 不可被继承 final 类
    : "enum" 枚举类
    : "open" 可继承 open 类
    : "annotation" 注解类
    : "sealed" 密封类
    : "data" 数据类
    ;

memberModifier
    : "override" 重写函数
    : "open" 可被重写
    : "final" 不可被重写
    : "abstract" 抽象函数
    : "lateinit" 后期初始化
    ;

accessModifier 访问权限控制，默认是 public
    : "private"
    : "protected"
    : "public"
    : "internal"
    ;

varianceAnnotation 泛型可变性
    : "in"
    : "out"
    ;

parameterModifier
    : "noinline"
    : "crossinline"
    : "vararg" 变长参数
    ;

typeParameterModifier
    : "reified"
    ;

functionModifier
    : "tailrec" 尾递归
    : "operator"
    : "infix"
    : "inline"
    : "external"
```

```
    : suspendModifier
    ;

propertyModifier
    : "const"
    ;

suspendModifier
    : "suspend"
    ;
```

其中，internel 访问权限，指的是在整个模块内可访问。模块（module）是指一起编译的一组 Kotlin 源代码文件：例如，IntelliJ IDEA 模块、Maven 项目或 Gradle 项目，通过 Ant 任务的一次调用编译的一组文件等。

这些修饰符的完整定义，在 kotlin/compiler/frontend/src/org/jetbrains/kotlin/lexer/KtTokens.java 源码中：

```
KtModifierKeywordToken[] MODIFIER_KEYWORDS_ARRAY =
            new KtModifierKeywordToken[] {
                    ABSTRACT_KEYWORD, ENUM_KEYWORD, OPEN_KEYWORD, INNER_KEYWORD,
                        OVERRIDE_KEYWORD, PRIVATE_KEYWORD,
                    PUBLIC_KEYWORD, INTERNAL_KEYWORD, PROTECTED_KEYWORD, OUT_
                        KEYWORD, IN_KEYWORD, FINAL_KEYWORD, VARARG_KEYWORD,
                    REIFIED_KEYWORD, COMPANION_KEYWORD, SEALED_KEYWORD, LATEINIT_
                        KEYWORD,
                    DATA_KEYWORD, INLINE_KEYWORD, NOINLINE_KEYWORD, TAILREC_
                        KEYWORD, EXTERNAL_KEYWORD, ANNOTATION_KEYWORD, CROSSINLINE_
                        KEYWORD,
                    CONST_KEYWORD, OPERATOR_KEYWORD, INFIX_KEYWORD, SUSPEND_
                        KEYWORD, HEADER_KEYWORD, IMPL_KEYWORD
            };

    TokenSet MODIFIER_KEYWORDS = TokenSet.create(MODIFIER_KEYWORDS_ARRAY);

    TokenSet TYPE_MODIFIER_KEYWORDS = TokenSet.create(SUSPEND_KEYWORD);
    TokenSet TYPE_ARGUMENT_MODIFIER_KEYWORDS = TokenSet.create(IN_KEYWORD,
        OUT_KEYWORD);
    TokenSet RESERVED_VALUE_PARAMETER_MODIFIER_KEYWORDS = TokenSet.create(OUT_
        KEYWORD, VARARG_KEYWORD);

    TokenSet VISIBILITY_MODIFIERS = TokenSet.create(PRIVATE_KEYWORD, PUBLIC_
        KEYWORD, INTERNAL_KEYWORD, PROTECTED_KEYWORD);
```

3.7.2　关键字（保留字）

每一门语言都自己的一套关键字集合。在 Kotlin 中，这些关键字定义如下：

```
TokenSet KEYWORDS = TokenSet.create(PACKAGE_KEYWORD, AS_KEYWORD, TYPE_ALIAS_
    KEYWORD, CLASS_KEYWORD, INTERFACE_KEYWORD,
```

```
    THIS_KEYWORD, SUPER_KEYWORD, VAL_KEYWORD, VAR_KEYWORD, FUN_KEYWORD, FOR_
        KEYWORD,
    NULL_KEYWORD,
    TRUE_KEYWORD, FALSE_KEYWORD, IS_KEYWORD,
    IN_KEYWORD, THROW_KEYWORD, RETURN_KEYWORD, BREAK_KEYWORD, CONTINUE_KEYWORD,
        OBJECT_KEYWORD, IF_KEYWORD,
    ELSE_KEYWORD, WHILE_KEYWORD, DO_KEYWORD, TRY_KEYWORD, WHEN_KEYWORD,
    NOT_IN, NOT_IS, AS_SAFE,
    TYPEOF_KEYWORD
    );

    TokenSet SOFT_KEYWORDS = TokenSet.create(FILE_KEYWORD, IMPORT_KEYWORD, WHERE_
        KEYWORD, BY_KEYWORD, GET_KEYWORD,
    SET_KEYWORD, ABSTRACT_KEYWORD, ENUM_KEYWORD, OPEN_KEYWORD, INNER_KEYWORD,
    OVERRIDE_KEYWORD, PRIVATE_KEYWORD, PUBLIC_KEYWORD, INTERNAL_KEYWORD, PROTECTED_
        KEYWORD,
    CATCH_KEYWORD, FINALLY_KEYWORD, OUT_KEYWORD, FINAL_KEYWORD, VARARG_KEYWORD,
        REIFIED_KEYWORD,
    DYNAMIC_KEYWORD, COMPANION_KEYWORD, CONSTRUCTOR_KEYWORD, INIT_KEYWORD, SEALED_
        KEYWORD,
    FIELD_KEYWORD, PROPERTY_KEYWORD, RECEIVER_KEYWORD, PARAM_KEYWORD, SETPARAM_KEYWORD,
    DELEGATE_KEYWORD,
    LATEINIT_KEYWORD,
    DATA_KEYWORD, INLINE_KEYWORD, NOINLINE_KEYWORD, TAILREC_KEYWORD, EXTERNAL_KEYWORD,
    ANNOTATION_KEYWORD, CROSSINLINE_KEYWORD, CONST_KEYWORD, OPERATOR_KEYWORD, INFIX_
        KEYWORD,
    SUSPEND_KEYWORD, HEADER_KEYWORD, IMPL_KEYWORD
    );
```

其中，对应的关键字如下：

```
KtKeywordToken PACKAGE_KEYWORD          = KtKeywordToken.keyword("package");
KtKeywordToken AS_KEYWORD               = KtKeywordToken.keyword("as");
KtKeywordToken TYPE_ALIAS_KEYWORD       = KtKeywordToken.keyword("typealias");
KtKeywordToken CLASS_KEYWORD            = KtKeywordToken.keyword("class");
KtKeywordToken THIS_KEYWORD             = KtKeywordToken.keyword("this");
KtKeywordToken SUPER_KEYWORD            = KtKeywordToken.keyword("super");
KtKeywordToken VAL_KEYWORD              = KtKeywordToken.keyword("val");
KtKeywordToken VAR_KEYWORD              = KtKeywordToken.keyword("var");
KtKeywordToken FUN_KEYWORD              = KtKeywordToken.keyword("fun");
KtKeywordToken FOR_KEYWORD              = KtKeywordToken.keyword("for");
KtKeywordToken NULL_KEYWORD             = KtKeywordToken.keyword("null");
KtKeywordToken TRUE_KEYWORD             = KtKeywordToken.keyword("true");
KtKeywordToken FALSE_KEYWORD            = KtKeywordToken.keyword("false");
KtKeywordToken IS_KEYWORD               = KtKeywordToken.keyword("is");
KtModifierKeywordToken IN_KEYWORD       = KtModifierKeywordToken.keywordModifier("in");
KtKeywordToken THROW_KEYWORD            = KtKeywordToken.keyword("throw");
KtKeywordToken RETURN_KEYWORD           = KtKeywordToken.keyword("return");
KtKeywordToken BREAK_KEYWORD            = KtKeywordToken.keyword("break");
KtKeywordToken CONTINUE_KEYWORD         = KtKeywordToken.keyword("continue");
KtKeywordToken OBJECT_KEYWORD           = KtKeywordToken.keyword("object");
```

```
KtKeywordToken IF_KEYWORD                = KtKeywordToken.keyword("if");
KtKeywordToken TRY_KEYWORD               = KtKeywordToken.keyword("try");
KtKeywordToken ELSE_KEYWORD              = KtKeywordToken.keyword("else");
KtKeywordToken WHILE_KEYWORD             = KtKeywordToken.keyword("while");
KtKeywordToken DO_KEYWORD                = KtKeywordToken.keyword("do");
KtKeywordToken WHEN_KEYWORD              = KtKeywordToken.keyword("when");
KtKeywordToken INTERFACE_KEYWORD         = KtKeywordToken.keyword("interface");


KtKeywordToken TYPEOF_KEYWORD            = KtKeywordToken.keyword("typeof");
...
KtKeywordToken FILE_KEYWORD      = KtKeywordToken.softKeyword("file");
KtKeywordToken FIELD_KEYWORD     = KtKeywordToken.softKeyword("field");
KtKeywordToken PROPERTY_KEYWORD     = KtKeywordToken.softKeyword("property");
KtKeywordToken RECEIVER_KEYWORD     = KtKeywordToken.softKeyword("receiver");
KtKeywordToken PARAM_KEYWORD     = KtKeywordToken.softKeyword("param");
KtKeywordToken SETPARAM_KEYWORD  = KtKeywordToken.softKeyword("setparam");
KtKeywordToken DELEGATE_KEYWORD  = KtKeywordToken.softKeyword("delegate");
KtKeywordToken IMPORT_KEYWORD    = KtKeywordToken.softKeyword("import");
KtKeywordToken WHERE_KEYWORD     = KtKeywordToken.softKeyword("where");
KtKeywordToken BY_KEYWORD        = KtKeywordToken.softKeyword("by");
KtKeywordToken GET_KEYWORD       = KtKeywordToken.softKeyword("get");
KtKeywordToken SET_KEYWORD       = KtKeywordToken.softKeyword("set");
KtKeywordToken CONSTRUCTOR_KEYWORD = KtKeywordToken.softKeyword("constructor");
KtKeywordToken INIT_KEYWORD        = KtKeywordToken.softKeyword("init");

KtModifierKeywordToken ABSTRACT_KEYWORD  = KtModifierKeywordToken.softKeywordModifier("abstract");
KtModifierKeywordToken ENUM_KEYWORD      = KtModifierKeywordToken.softKeywordModifier("enum");
KtModifierKeywordToken OPEN_KEYWORD      = KtModifierKeywordToken.softKeywordModifier("open");
KtModifierKeywordToken INNER_KEYWORD     = KtModifierKeywordToken.softKeywordModifier("inner");
KtModifierKeywordToken OVERRIDE_KEYWORD  = KtModifierKeywordToken.softKeywordModifier("override");
KtModifierKeywordToken PRIVATE_KEYWORD   = KtModifierKeywordToken.softKeywordModifier("private");
KtModifierKeywordToken PUBLIC_KEYWORD    = KtModifierKeywordToken.softKeywordModifier("public");
KtModifierKeywordToken INTERNAL_KEYWORD  = KtModifierKeywordToken.softKeywordModifier("internal");
KtModifierKeywordToken PROTECTED_KEYWORD = KtModifierKeywordToken.softKeywordModifier("protected");
KtKeywordToken CATCH_KEYWORD     = KtKeywordToken.softKeyword("catch");
KtModifierKeywordToken OUT_KEYWORD       = KtModifierKeywordToken.softKeywordModifier("out");
KtModifierKeywordToken VARARG_KEYWORD    = KtModifierKeywordToken.softKeywordModifier("vararg");
KtModifierKeywordToken REIFIED_KEYWORD   = KtModifierKeywordToken.softKeywordModifier("reified");
KtKeywordToken DYNAMIC_KEYWORD   = KtKeywordToken.softKeyword("dynamic");
KtModifierKeywordToken COMPANION_KEYWORD = KtModifierKeywordToken.softKeywordModifier("companion");
KtModifierKeywordToken SEALED_KEYWORD    = KtModifierKeywordToken.softKeywordModifier("sealed");

KtModifierKeywordToken DEFAULT_VISIBILITY_KEYWORD = PUBLIC_KEYWORD;

KtKeywordToken FINALLY_KEYWORD   = KtKeywordToken.softKeyword("finally");
KtModifierKeywordToken FINAL_KEYWORD     = KtModifierKeywordToken.softKeywordModifier("final");

KtModifierKeywordToken LATEINIT_KEYWORD  = KtModifierKeywordToken.softKeywordModifier("lateinit");

KtModifierKeywordToken DATA_KEYWORD       = KtModifierKeywordToken.softKeywordModifier("data");
```

```
KtModifierKeywordToken INLINE_KEYWORD      = KtModifierKeywordToken.softKeywordModifier("inline");
KtModifierKeywordToken NOINLINE_KEYWORD    = KtModifierKeywordToken.softKeywordModifier("noinline");
KtModifierKeywordToken TAILREC_KEYWORD     = KtModifierKeywordToken.softKeywordModifier("tailrec");
KtModifierKeywordToken EXTERNAL_KEYWORD    = KtModifierKeywordToken.softKeywordModifier("external");
KtModifierKeywordToken ANNOTATION_KEYWORD = KtModifierKeywordToken.softKeywordModifier("annotation");
KtModifierKeywordToken CROSSINLINE_KEYWORD = KtModifierKeywordToken.softKeywordModifier("crossinline");
KtModifierKeywordToken OPERATOR_KEYWORD = KtModifierKeywordToken.softKeywordModifier("operator");
KtModifierKeywordToken INFIX_KEYWORD     = KtModifierKeywordToken.softKeywordModifier("infix");

KtModifierKeywordToken CONST_KEYWORD     = KtModifierKeywordToken.softKeywordModifier("const");

KtModifierKeywordToken SUSPEND_KEYWORD  = KtModifierKeywordToken.softKeywordModifier("suspend");

KtModifierKeywordToken HEADER_KEYWORD    = KtModifierKeywordToken.softKeywordModifier("header");
KtModifierKeywordToken IMPL_KEYWORD      = KtModifierKeywordToken.softKeywordModifier("impl");
```

1. this 关键字

this 关键字持有当前对象的引用。可以使用 this 来引用变量或者成员函数，也可以使用 return this 来返回某个类的引用。

代码示例如下：

```
class ThisDemo {
    val thisis = "THIS IS"

    fun whatIsThis(): ThisDemo {
        println(this.thisis)        //引用变量
        this.howIsThis()            //引用成员函数
        return this                 //返回此类的引用
    }

    fun howIsThis(){
        println("HOW IS THIS ?")
    }
}
```

测试代码：

```
@Test
fun testThisDemo(){
    val demo = ThisDemo()
    println(demo.whatIsThis())
}
```

输出：

```
THIS IS
HOW IS THIS ?
com.easy.kotlin.ThisDemo@475232fc
```

在类的成员中，this 指向的是该类的当前对象。

在扩展函数或者带接收者的函数字面值中，this 表示在点左侧传递的接收者参数。

代码示例如下：

```
>>> val sum = fun Int.(x:Int):Int = this + x
>>> sum
kotlin.Int.(kotlin.Int) -> kotlin.Int
>>> 1.sum(1)
2
>>> val concat = fun String.(x:Any) = this + x
>>> "abc".concat(123)
abc123
>>> "abc".concat(true)
abctrue
```

如果 this 没有限定符，它指的是最内层的包含它的作用域。如果我们想要引用其他作用域中的 this，可以使用 this@label 标签。

代码示例如下：

```
class Outer {
    val oh = "Oh!"

    inner class Inner {

        fun m() {
            val outer = this@Outer
            val inner = this@Inner
            val pthis = this
            println("outer=" + outer)
            println("inner=" + inner)
            println("pthis=" + pthis)
            println(this@Outer.oh)

            val fun1 = hello@ fun String.() {
                val d1 = this //fun1 的接收者
                println("d1" + d1)
            }

            val fun2 = { s: String ->
                val d2 = this
                println("d2=" + d2)
            }

            "abc".fun1()

            fun2

        }
    }
}
```

测试代码：

```
@Test
fun testThisKeyWord() {
    val outer = Outer()
    outer.Inner().m()
}
```

输出为：

```
outer=com.easy.kotlin.Outer@5114e183
inner=com.easy.kotlin.Outer$Inner@5aa8ac7f
pthis=com.easy.kotlin.Outer$Inner@5aa8ac7f
Oh!
d1abc
```

2. super 关键字

super 关键字持有指向其父类的引用。

代码示例如下：

```
open class Father {
    open val firstName = "Chen"
    open val lastName = "Jason"

    fun ff() {
        println("FFF")
    }
}

class Son : Father {
    override var firstName = super.firstName
    override var lastName = "Jack"

    constructor(lastName: String) {
        this.lastName = lastName
    }

    fun love() {
        super.ff() //调用父类方法
        println(super.firstName + " " + super.lastName + " Love " + this.
            firstName + " " + this.lastName)
    }
}
```

测试代码：

```
@Test
fun testSuperKeyWord() {
    val son = Son("Harry")
    son.love()
}
```

输出为：

```
FFF
Chen Jason Love Chen Harry
```

3.7.3 操作符和操作符的重载

Kotlin允许我们为自己的类型提供预定义的一组操作符的实现。这些操作符具有固定的符号表示（如+或*）和固定的优先级。这些操作符的符号定义如下：

```
KtSingleValueToken LBRACKET          = new KtSingleValueToken("LBRACKET", "[");
KtSingleValueToken RBRACKET          = new KtSingleValueToken("RBRACKET", "]");
KtSingleValueToken LBRACE            = new KtSingleValueToken("LBRACE", "{");
KtSingleValueToken RBRACE            = new KtSingleValueToken("RBRACE", "}");
KtSingleValueToken LPAR              = new KtSingleValueToken("LPAR", "(");
KtSingleValueToken RPAR              = new KtSingleValueToken("RPAR", ")");
KtSingleValueToken DOT               = new KtSingleValueToken("DOT", ".");
KtSingleValueToken PLUSPLUS          = new KtSingleValueToken("PLUSPLUS", "++");
KtSingleValueToken MINUSMINUS        = new KtSingleValueToken("MINUSMINUS", "--");
KtSingleValueToken MUL               = new KtSingleValueToken("MUL", "*");
KtSingleValueToken PLUS              = new KtSingleValueToken("PLUS", "+");
KtSingleValueToken MINUS             = new KtSingleValueToken("MINUS", "-");
KtSingleValueToken EXCL              = new KtSingleValueToken("EXCL", "!");
KtSingleValueToken DIV               = new KtSingleValueToken("DIV", "/");
KtSingleValueToken PERC              = new KtSingleValueToken("PERC", "%");
KtSingleValueToken LT                = new KtSingleValueToken("LT", "<");
KtSingleValueToken GT                = new KtSingleValueToken("GT", ">");
KtSingleValueToken LTEQ              = new KtSingleValueToken("LTEQ", "<=");
KtSingleValueToken GTEQ              = new KtSingleValueToken("GTEQ", ">=");
KtSingleValueToken EQEQEQ            = new KtSingleValueToken("EQEQEQ", "===");
KtSingleValueToken ARROW             = new KtSingleValueToken("ARROW", "->");
KtSingleValueToken DOUBLE_ARROW      = new KtSingleValueToken("DOUBLE_ARROW", "=>");
KtSingleValueToken EXCLEQEQEQ        = new KtSingleValueToken("EXCLEQEQEQ", "!==");
KtSingleValueToken EQEQ              = new KtSingleValueToken("EQEQ", "==");
KtSingleValueToken EXCLEQ            = new KtSingleValueToken("EXCLEQ", "!=");
KtSingleValueToken EXCLEXCL          = new KtSingleValueToken("EXCLEXCL", "!!");
KtSingleValueToken ANDAND            = new KtSingleValueToken("ANDAND", "&&");
KtSingleValueToken OROR              = new KtSingleValueToken("OROR", "||");
KtSingleValueToken SAFE_ACCESS       = new KtSingleValueToken("SAFE_ACCESS", "?.");
KtSingleValueToken ELVIS             = new KtSingleValueToken("ELVIS", "?:");
KtSingleValueToken QUEST             = new KtSingleValueToken("QUEST", "?");
KtSingleValueToken COLONCOLON        = new KtSingleValueToken("COLONCOLON", "::");
KtSingleValueToken COLON             = new KtSingleValueToken("COLON", ":");
KtSingleValueToken SEMICOLON         = new KtSingleValueToken("SEMICOLON", ";");
KtSingleValueToken DOUBLE_SEMICOLON = new KtSingleValueToken("DOUBLE_SEMICOLON", ";;");
KtSingleValueToken RANGE             = new KtSingleValueToken("RANGE", "..");
KtSingleValueToken EQ                = new KtSingleValueToken("EQ", "=");
KtSingleValueToken MULTEQ            = new KtSingleValueToken("MULTEQ", "*=");
KtSingleValueToken DIVEQ             = new KtSingleValueToken("DIVEQ", "/=");
KtSingleValueToken PERCEQ            = new KtSingleValueToken("PERCEQ", "%=");
```

```
KtSingleValueToken PLUSEQ          = new KtSingleValueToken("PLUSEQ", "+=");
KtSingleValueToken MINUSEQ         = new KtSingleValueToken("MINUSEQ", "-=");
KtKeywordToken NOT_IN              = KtKeywordToken.keyword("NOT_IN", "!in");
KtKeywordToken NOT_IS              = KtKeywordToken.keyword("NOT_IS", "!is");
KtSingleValueToken HASH            = new KtSingleValueToken("HASH", "#");
KtSingleValueToken AT              = new KtSingleValueToken("AT", "@");

KtSingleValueToken COMMA           = new KtSingleValueToken("COMMA", ",");
```

3.7.4 操作符优先级

Kotlin 操作符的优先级（Precedence）如表 3-1 所示。

表 3-1 操作符优先级

优先级	标 题	符 号
最高	后缀（Postfix）	++, --, ., ?., ?
	前缀（Prefix）	-, +, ++, --, !, labelDefinition@
	右手类型运算（Type RHS，right-hand side class type (RHS)）	:, as, as?
	乘除取余（Multiplicative）	*, /, %
	加减（Additive）	+, -
	区间范围（Range）	..
	Infix 函数	例如，给 Int 定义扩展 infix fun Int.shl(x: Int): Int {...}，这样调用 1 shl 2，等同于 1.shl(2)
	Elvis 操作符	?:
	命名检查符（Named checks）	in, !in, is, !is
	比较大小（Comparison）	<, >, <=, >=
	相等性判断（Equality）	==, !=, ===, !==
	与（Conjunction）	&&
	或（Disjunction）	||
最低	赋值（Assignment）	=, +=, -=, *=, /=, %=

为实现这些的操作符，Kotlin 为二元操作符左侧的类型和一元操作符的参数类型，提供了相应的函数或扩展函数。

例如在 kotlin/core/builtins/native/kotlin/Primitives.kt 代码中，对基本类型 Int 的操作符的实现代码如下：

```
public class Int private constructor() : Number(), Comparable<Int> {
    ...
    public operator fun compareTo(other: Byte): Int
    public operator fun compareTo(other: Short): Int
    public override operator fun compareTo(other: Int): Int
    public operator fun compareTo(other: Long): Int
```

```
    public operator fun compareTo(other: Float): Int
    public operator fun compareTo(other: Double): Int
    public operator fun plus(other: Byte): Int
    public operator fun plus(other: Short): Int
    public operator fun plus(other: Int): Int/
    public operator fun plus(other: Long): Long
    public operator fun plus(other: Float): Float
    public operator fun plus(other: Double): Double
    public operator fun minus(other: Byte): Int
    public operator fun minus(other: Short): Int
    public operator fun minus(other: Int): Int
    public operator fun minus(other: Long): Long
    public operator fun minus(other: Float): Float
    public operator fun minus(other: Double): Double
    ...
    public override fun toByte(): Byte
    public override fun toChar(): Char
    public override fun toShort(): Short
    public override fun toInt(): Int
    public override fun toLong(): Long
    public override fun toFloat(): Float
    public override fun toDouble(): Double
}
```

从源代码我们可以看出，重载操作符的函数需要用 operator 修饰符标记。中缀操作符的函数使用 infix 修饰符标记。

3.7.5　一元操作符

本小节我们介绍 Kotlin 的一元操作符（unary operation），包括前缀、递增和递减。

1. 前缀操作符

表达式	翻译为
+a	a.unaryPlus()
−a	a.unaryMinus()
!a	a.not()

例如，当编译器处理表达式 +a 时，将执行以下步骤：

1）确定 a 的类型，令其为 T。

2）为接收者 T 查找一个带有 operator 修饰符的无参函数 unaryPlus()，即成员函数或扩展函数。

3）如果函数不存在或不明确，则导致编译错误。

4）如果函数存在且其返回类型为 R，则表达式 +a 具有类型 R。

编译器对这些操作以及所有其他操作都针对基本类型做了优化，不会引入函数调用的开销。

以下是如何重载一元减运算符的示例：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/6/10.
 */

class OperatorDemo {

}


data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)
```

测试代码：

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4::class)
class OperatorDemoTest {

    @Test
    fun testPointUnaryMinus() {
        val p = Point(1, 1)
        val np = -p
        println(np) //Point(x=-1, y=-1)
    }
}
```

2. 递增和递减

表达式	翻译为
a++	a.inc() 返回值是 a
a--	a.dec() 返回值是 a
++a	a.inc() 返回值是 a+1
--a	a.dec() 返回值是 a-1

inc() 和 dec() 函数必须返回一个值，它用于赋值给使用 ++ 或 -- 操作的变量。

编译器解析后缀形式的操作符，例如 a++，步骤如下：

1）确定 a 的类型，令其为 T。

2）查找一个适用于类型为 T 的接收者的、带有 operator 修饰符的无参数函数 inc()。

3）检查函数的返回类型是 T 的子类型。

计算表达式的步骤是：

1）把 a 的初始值存储到临时存储 a_ 中。

2）把 a.inc() 结果赋值给 a。

3）把 a_ 作为表达式的结果返回。

(a-- 同理分析。)

对于前缀形式 ++a 和 --a 解析步骤类似，但是返回值是取新值来返回：

1）把 a.inc() 结果赋值给 a。

2）把 a 的新值 a+1 作为表达式结果返回。

(--a 同理分析。)

3.7.6　二元操作符

1. 算术运算符

表达式	翻译为
a + b	a.plus(b)
a – b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)、a.mod(b)
a..b	a.rangeTo(b)

代码示例：

```
>>> val a=10
>>> val b=3
>>> a+b
13
>>> a-b
7
>>> a/b
3
>>> a%b
1
>>> a..b
10..3
>>> b..a
3..10
```

2. 字符串的 + 运算符重载

先用代码举个例子：

```
>>> ""+1
```

```
1
>>> 1+""
error: none of the following functions can be called with the arguments supplied:
public final operator fun plus(other: Byte): Int defined in kotlin.Int
public final operator fun plus(other: Double): Double defined in kotlin.Int
public final operator fun plus(other: Float): Float defined in kotlin.Int
public final operator fun plus(other: Int): Int defined in kotlin.Int
public final operator fun plus(other: Long): Long defined in kotlin.Int
public final operator fun plus(other: Short): Int defined in kotlin.Int
1+""
 ^
```

从上面的示例，我们可以看出，在 Kotlin 中 1+"" 是不允许的（相比 Scala，写这样的 Kotlin 代码就显得不大友好），只能显式调用 toString 来相加：

```
>>> 1.toString()+""
1
```

3. 自定义重载的 + 运算符

下面我们使用一个计数类 Counter 重载的 + 运算符来增加 index 的计数值。

代码示例：

```
data class Counter(var index: Int)

operator fun Counter.plus(increment: Int): Counter {
    return Counter(index + increment)
}
```

测试类：

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

/**
 * Created by jack on 2017/6/10.
 */
@RunWith(JUnit4::class)
class OperatorDemoTest
    @Test
    fun testCounterIndexPlus() {
        val c = Counter(1)
        val cplus = c + 10
        println(cplus) //Counter(index=11)
    }
}
```

4. in 操作符

表达式	翻译为
a in b	b.contains(a)
a !in b	!b.contains(a)

这里的 in 操作符转换为 contains 函数。

5. 索引访问操作符

表达式	翻译为
a[i]	a.get(i)
a[i] = b	a.set(i, b)

方括号转换为调用带有适当数量参数的 get 和 set。

6. 调用操作符

表达式	翻译为
a()	a.invoke()
a(i)	a.invoke(i)

圆括号转换为调用带有适当数量参数的 invoke。

7. 计算并赋值

表达式	翻译为
a += b	a.plusAssign(b)
a −= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.modAssign(b)

对于赋值操作，例如 a += b，编译器会试着生成 a = a + b 的代码（这里包含类型检查：a + b 的类型必须是 a 的子类型）。

8. 相等与不等操作符

Kotlin 中有两种类型的相等性：

- 引用相等 === !==（两个引用指向同一对象）
- 结构相等 == !=（使用 equals() 判断）

表达式	翻译为
a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null))

== 操作符有些特殊：它被翻译成一个复杂的表达式，用于筛选 null 值。意思是：如果 a 不是 null 则调用 equals(Any?) 函数并返回其值；否则（即 a === null）就计算 b === null 的

值并返回。

当与 null 显式比较时，a == null 会被自动转换为 a=== null。



=== 和 !== 不可重载。

9. Elvis 操作符 ?:

在 Kotin 中，Elvis 操作符特定是跟 null 比较，主要用来作 null 安全性检查。也就是说：

```
y = x?:0
```

等价于：

```
val y = if(x!==null) x else 0
```

Elvis 操作符 ?: 是一个二元运算符，如果第一个操作数为真，则返回第一个操作数，否则将计算并返回其第二个操作数。它是三元条件运算符的变体。命名灵感来自猫王的发型风格。

Kotlin 中没有这样的三元运算符 true?1:0，取而代之的是 if(true) 1 else 0。而 Elvis 操作符算是精简版的三元运算符。

我们在 Java 中使用的三元运算符的语法，通常要重复变量两次，示例如下：

```
String name = "Elvis Presley";
String displayName = (name != null) ? name : "Unknown";
```

可以使用 Elvis 操作符作同样操作：

```
String name = "Elvis Presley";
String displayName = name?:"Unknown"
```

我们可以看出，用 Elvis 操作符（?:）可以把带有默认值的 if/else 结构写得及其短小。用 Elvis 操作符不用检查 null（避免了 NullPointerException），也不用重复变量。

Elvis 操作符功能在 Spring 表达式语言（SpEL）中提供。Kotlin 当然就没有理由不支持这个特性。

代码示例：

```
>>> val x = null
>>> val y = x?:0
>>> y
0
>>> val x = false
>>> val y = x?:0
>>> y
false
>>> val x = ""
```

```
>>> val y = x?:0
>>> y

>>> val x = "abc"
>>> val y = x?:0
>>> y
abc
```

10. 比较操作符

表达式	翻译为
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

所有的比较都转换为对 compareTo 的调用，这个函数需要返回 Int 值。

11. 用 infix 函数自定义中缀操作符

我们可以通过自定义 infix 函数来实现中缀操作符。

代码示例：

```
data class Person(val name: String, val age: Int)

infix fun Person.grow(years: Int): Person {
    return Person(name, age + years)
}
```

测试代码：

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

/**
 * Created by jack on 2017/6/11.
 */
@RunWith(JUnit4::class)
class InfixFunctionDemoTest {

    @Test fun testInfixFuntion() {
        val person = Person("Jack", 20)

        println(person.grow(2))

        println(person grow 2)
    }
}
```

输出为：

```
Person(name=Jack, age=22)
Person(name=Jack, age=22)
```

3.8 扩展函数和扩展属性

Kotlin 支持“扩展函数”和“扩展属性”。能够扩展一个类的新功能而无需继承该类或使用像装饰者这样的设计模式等。

大多数时候我们在顶层定义扩展，即直接在包里定义：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/6/11.
 */

val <T> List<T>.lastIndex: Int get() = size - 1

fun String.notEmpty(): Boolean {
    return !this.isEmpty()
}
```

这样我们就可以在整个包里使用这些扩展。

要使用其他包的扩展，需要在调用方导入它：

```
package com.example.usage

import foo.bar.goo //导入所有名为“goo”的扩展
                   //或者
import foo.bar.*   //从“foo.bar”导入一切

fun usage(baz: Baz) {
    baz.goo()
}
```

3.8.1 扩展函数

声明一个扩展函数需要用“被扩展的类型”来作为前缀。比如，我们不喜欢类似下面的“双重否定”式的逻辑判断（绕脑子）：

```
>>> !"123".isEmpty()
true
```

我们就可以为 String 类型扩展一个 notEmpty() 函数：

```
>>> fun String.notEmpty():Boolean{
... return !this.isEmpty()
```

```
... }

>>> "".notEmpty()
false

>>> "123".notEmpty()
true
```

下面代码为MutableList<Int>添加一个swap函数：

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] //this对应该列表
    this[index1] = this[index2]
    this[index2] = tmp
}
```

this关键字在扩展函数内部对应到接收者对象（传过来的在点.符号前的对象）。现在，我们对任意MutableList<Int>调用该函数了。

当然，这个函数对任何MutableList<T>起作用，我们可以泛化它：

```
fun <T> MutableList<T>.mswap(index1: Int, index2: Int) {
    val tmp = this[index1] //“this”对应该列表
    this[index1] = this[index2]
    this[index2] = tmp
}
```

为了在接收者类型表达式中使用泛型，我们要在函数名前声明泛型参数。

完整代码示例如下：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/6/11.
 */

val <T> List<T>.lastIndex: Int get() = size - 1

fun String.notEmpty(): Boolean {
    return !this.isEmpty()
}

fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] //this对应该列表m
    this[index1] = this[index2]
    this[index2] = tmp
}

fun <T> MutableList<T>.mswap(index1: Int, index2: Int) {
    val tmp = this[index1] //“this”对应该列表
    this[index1] = this[index2]
    this[index2] = tmp
```

```
}

class ExtensionsDemo {


    fun useExtensions() {
        val a = "abc"
        println(a.notEmpty())//true

        val mList = mutableListOf<Int>(1, 2, 3, 4, 5)
        println("Before Swap:")
        println(mList)//[1, 2, 3, 4, 5]
        mList.swap(0, mList.size - 1)
        println("After Swap:")
        println(mList)//[5, 2, 3, 4, 1]


        val mmList = mutableListOf<String>("a12", "b34", "c56", "d78")
        println("Before Swap:")
        println(mmList)//[a12, b34, c56, d78]
        mmList.mswap(1, 2)
        println("After Swap:")
        println(mmList)//[a12, c56, b34, d78]


        val mmmList = mutableListOf<Int>(100, 200, 300, 400, 500)
        println("Before Swap:")
        println(mmmList)
        mmmList.mswap(0, mmmList.lastIndex)
        println("After Swap:")
        println(mmmList)
    }


    class Inner {
        fun useExtensions() {
            val mmmList = mutableListOf<Int>(100, 200, 300, 400, 500)
            println(mmmList.lastIndex)
        }
    }
}
```

测试代码：

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

/**
```

```
 * Created by jack on 2017/6/11.
 */

@RunWith(JUnit4::class)
class ExtensionsDemoTest {
    @Test fun testExtensionsDemo() {
        val demo = ExtensionsDemo()
        demo.useExtensions()
    }
}
```

扩展不是真正的修改所扩展的类。我们定义一个扩展，其实并没有在一个类中插入新函数，仅仅是通过该类型的变量，用点（.）表达式去调用这个新函数。

3.8.2　扩展属性

和函数类似，Kotlin 支持扩展属性，如下代码所示：

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

注意，由于扩展没有实际地将成员插入类中，因此对扩展的属性来说，它的行为只能由显式提供的 getters/setters 定义。

代码示例如下：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/6/11.
 */

val <T> List<T>.lastIndex: Int get() = size - 1
```

我们可以直接使用包 com.easy.kotlin 中扩展的属性 lastIndex，如图 3-4 所示。

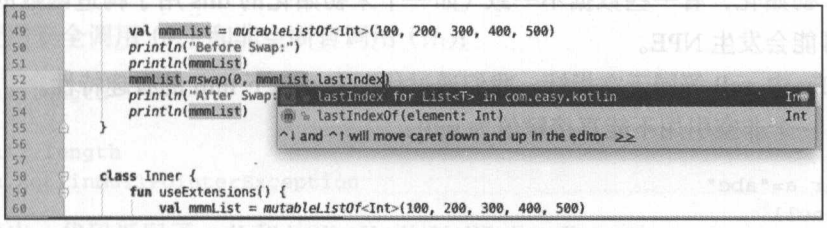


图 3-4　包 com.easy.kotlin 中扩展的属性 lastIndex

3.9　空指针安全

Java 最大的一个问题就是 null。我们写代码的时候知道，在 Java 中 NPE（NullPointer-

Exceptions）是一件令程序员几近崩溃的事情。如果没有对变量或参数进行 null 判断，那么程序当中就有可能抛出大量的 NullPointerException，然而在编码时这些又是难以检测到的。很多时候，虽然费尽体力脑力，仍然防不胜防。以前，当我们不确定一个 DTO 类中的字段是否已初始化时，可以使用 @Nullable 和 @NotNull 注解来声明，但功能仍然很有限。

我们通常把 null 理解为编程语言中定义特殊的 0，把我们初始化的指针指向它，以防止“野指针”的恶果。在 Java 中，null 是任何引用类型的默认值，不严格的说是所有 Object 类型的默认值。

这里的 null 既不是对象也不是一种类型，它仅是一种特殊的值，我们可以将其赋予任何引用类型，也可以将 null 转化成任何类型。在编译和运行时期，将 null 强制转换成任何引用类型都是可行的，在运行时期都不会抛出空指针异常。注意，这里指的是任何 Java 的引用类型。在遇到基本类型 int long float double short byte 等的时候，情况就不一样了。而且还是个坑。编译器不会报错，但是运行时会抛 NPE。空指针异常。这是 Java 中的自动拆箱导致的。代码示例：

```
Integer nullInt = null;    //this is ok
int anotherInt = nullInt; //编译器允许这么赋值，但是在运行时抛 NullPointerException
```

所以，我们写 Java 代码的时候，要时刻注意这一点：Integer 的默认值是 null 而不是 0。当把 null 值传递给一个 int 型变量的时候，Java 的自动装箱将会返回空指针异常。

Kotlin 针对 Java 的这个问题作了改进。它使用了显式的 null，这会强制我们在必要时进行 null 检查。Kotlin 通过类型系统中的可空类型来实现空指针的安全检查。关于 Kotlin 的可空类型以及类型系统将在下章介绍。

但是，当我们的代码有如下情况时：

- ❑ 显式调用 throw NullPointerException()
- ❑ 使用了 !! 操作符
- ❑ 调用的外部 Java 代码有 NPE
- ❑ 对于初始化，有一些数据不一致（如一个未初始化的 this 用于构造函数的某个地方）也可能会发生 NPE。

在 Kotlin 中 null 等同于空指针。我们通过代码来看一下 null 的有趣特性。

首先，一个非空引用不能直接赋值为 null：

```
>>> var a="abc"
>>> a=null
error: null can not be a value of a non-null type String
a=null
  ^

>>> var one=1
>>> one=null
error: null can not be a value of a non-null type Int
```

```
one=null
    ^

>>> var arrayInts = intArrayOf(1,2,3)
>>> arrayInts=null
error: null can not be a value of a non-null type IntArray
arrayInts=null
          ^
```

这样，我们就可以放心地调用 a 的方法或者访问它的属性，不会导致 NPE：

```
>>> val a="abc"
>>> a.length
3
```

如果要允许为空，我们可以在变量的类型后面加个问号？声明一个变量为可空的：

```
>>> var a:String?="abc"
>>> a=null
>>> var one:Int?=1
>>> one=null
>>> var arrayInts:IntArray?=intArrayOf(1,2,3)
>>> arrayInts=null
>>> arrayInts
null
```

如果我们声明了一个可空 String? 类型变量 na，然后直接调用 length 属性，这将是不安全的。编译器会直接报错：

```
>>> var na:String?="abc"
>>> na=null
>>> na.length
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable
    receiver of type String?
na.length
  ^
```

我们使用安全调用（?.）和非空断言调用（!!.）：

```
>>> na?.length
null
>>> na!!.length
kotlin.KotlinNullPointerException
```

可以看出，代码返回了 null 和 kotlin.KotlinNullPointerException。

安全调用在链式调用中很有用。在调用链中如果任意一个属性（环节）为空，这个链式调用就会安全返回 null。

如果只对非空值执行某个操作，安全调用操作符可以与 let（以调用者的值作为参数来执行指定的函数块，并返回其结果）一起使用：

```
>>> val listWithNulls: List<String?> = listOf("A", "B",null)
>>> listWithNulls
[A, B, null]

>>> listWithNulls.forEach{
... it?.let{println(it)}
... }
A
B
```

3.10 标准库 API 简介

Kotlin 标准类库为日常工作提供了重要工具，例如：各种实用程序用于处理字符串和字符序列作为 JDK 类的扩展，方便地处理文件流、IO、线程等操作。本节简单介绍 Kotlin 的标准库（stdlib）。

Kotlin 的 Java 和 JS 共用的公共库如表 3-2 所示。

表 3-2 Java 和 JS 共用的公共库

包 名	功能说明
kotlin	核心函数和数据类型，支持所有平台（JVM、JavaScript 等）
kotlin.annotation	为 Kotlin 注解功能提供支持
kotlin.collections	集合类型，如 Iterable、Collection、List、Set、Map 等
kotlin.comparisons	帮助函数用于创建 Comparator 比较器实例
kotlin.coroutines.experimental(1.1)	支持协程，包括支持延迟序列（lazy sequence）等
kotlin.coroutines.experimental.intrinsics(1.1)	基于协程的 API 库的底层构建块
kotlin.experimental(1.1)	实验 API，将来版本可能会改变
kotlin.io	IO API 用于处理文件和流
kotlin.properties	代理 / 委托属性的标准实现，帮助函数实现自定义代理 / 委托
kotlin.ranges	范围 / 区间，数列 Progressions 和相关扩展功能
kotlin.reflect	Kotlin 反射的运行时 API
kotlin.sequences	序列类型表示延迟求值的集合、实例化序列和扩展函数
kotlin.text	处理文本和正则表达式的函数

Java 平台的 Kotlin 库如表 3-3 所示。

表 3-3 Java 平台的 Kotlin 库

包 名	功能说明
kotlin.concurrent	并发（concurrent）编程 Kotlin 函数库
kotlin.jvm	Java 平台特有的函数和注解
kotlin.reflect.full(1.1)	Kotlin 反射库的扩展 Extensions
kotlin.reflect.jvm	Kotlin 与 Java 反射的互操作运行时 API
kotlin.streams(1.1,JRE8)	处理 Java 8 流的 Kotlin 函数库
kotlin.system	与系统有关的 Kotlin 函数库

JavaScript 平台的 Kotlin 库如表 3-4 所示。

表 3-4　JavaScript 平台的 Kotlin 库

包　名	功能说明
kotlin.browser	在浏览器环境下访问顶层属性（如 document、window 等）
kotlin.dom	处理浏览器 DOM 的 Kotlin 函数库
kotlin.js	JavaScript 平台特有的一些函数和 API
org.khronos.webgl	WebGL API 的 Kotlin 包装器（wrappers）
org.w3c.dom	DOM API 的 Kotlin 包装器
org.w3c.dom.css	DOM CSS API 的 Kotlin 包装器
org.w3c.dom.events	DOM events API 的 Kotlin 包装器
org.w3c.dom.parsing	DOM parsing API 的 Kotlin 包装器
org.w3c.dom.svg	DOM SVG API 的 Kotlin 包装器
org.w3c.dom.url	DOM URL API 的 Kotlin 包装器
org.w3c.xhr	XMLHttpRequest API 的 Kotlin 包装器
org.w3c.files	W3C file API 的 Kotlin 包装器
org.w3c.notifications	Web Notifications API 的 Kotlin 包装器
org.w3c.performance	Navigation Timing API 的 Kotlin 包装器
org.w3c.workers	Web Workers API 的 Kotlin 包装器

提示：详细的 API 文档介绍可参考 http://kotlinlang.org/api/latest/jvm/stdlib/index.html。

3.11　本章小结

本章我们学习了 Kotlin 语言的基本词汇（关键字、标识符等）、句子（流程控制、表达式、操作符等）和基础语法。同时，学习了空指针安全、扩展函数与扩展属性等的语言特性。空指针安全可以说是 Kotlin 中的一大特色，因为它解决了 Java 中令人头疼的 NPE 问题。

另外，Kotlin 中的扩展函数相当实用，可以说是 Kotlin 开发利器。例如，在后面章节关于 Kotlin DSL 的学习中，我们将会看到 Anko、kotlinx.html 等 DSL 给我们带来的全新的编程体验。在集合类、文件 IO 类等中，Kotlin 就在 Java API 的基础扩展了大量的实用函数。当然，Kotlin 不仅仅是“Better Than Java”，在本书后面的内容中我们会学习到更多的内容。

我们将在下一章节介绍 Kotlin 的基本类型和类型系统。

本章示例源码：https://github.com/EasyKotlin/chapter3_kotlin_basics



Chapter 4 第 4 章

基本数据类型与类型系统

道生一，一生二，二生三，三生万物。

——老子《道德经》第四十二章

本章我们学习 Kotlin 语言的基本类型和类型系统。

我们会首先简单介绍类型的基本概念，然后具体介绍 Kotlin 的内置基本类型：数字、字符串、布尔、数组等。接着介绍 Kotlin 中引入的特殊的可空类型。最后，我们简单介绍了 Kotlin 中的类型推断与类型转换的相关内容。

下面就让我们正式开始吧。

4.1 什么是类型

一切皆是映射。

在计算机中，任何数值都是以一组比特（01）组成的，硬件无法区分内存地址、脚本、字符、整数以及浮点数。这个时候，我们使用类型赋予一组比特以特定的意义。

类型（type）本质上就是内存中的数值或变量对象的逻辑映射。

到目前为止，我们已经了解了 Kotlin 的基本符号以及基础语法。我们可以看出，使用 Kotlin 写的代码更简洁、可读性更好、更富有生产力。本章我们来学习一下 Kotlin 的基本数据类型与类型系统。

在计算机科学中，最早的类型系统用来区别数字里面的整数和浮点数。在 20 世纪五六十年代，这种分类扩展到了结构化的数据和高阶函数中。70 年代，引入了几个更为丰

富的概念，例如：参数化类型，抽象数据类型，模块系统，子类型等等，类型系统作为一个独立的领域形成了。

在每一门编程语言中，都有一个特定的类型系统（Type System）。类型系统是一门编程语言最核心也是最基础的部分。我们这里说的类型系统可以简单理解为以下两个部分：

- 一组基本类型构成的 PTS（Primary Type Set，基本类型集合）。
- PTS 上定义的一系列组合、运算、转换规则等。

这一简单优雅而惊人的世界构成观，贯穿了人类现实世界和计算机编程语言所定义的虚拟世界。或许语言的设计者也没有料想到，但是最终的结果确实是有限的设计导出了无限的可能性。

类型系统用于定义如何将编程语言中的数值和表达式归类为许多不同的类型，如何操作这些类型，这些类型如何互相作用等。

类型系统在各种语言之间有非常大的不同，主要的差异存在于编译时期的语法，以及运行时期的操作实现方式。

类型系统提供的主要功能有：安全性、最优化、可读性、抽象化。下面我们针对其中的三个功能解释一下。

1. 安全性

编译器可以使用类型来检查无意义的、或者是可能无效的代码。例如，在强类型的语言中，如果没有对字符串的 + 进行重载，那么如下表达式：

```
"Hello, World" + 3
```

就会被编译器检测出来，因为不能对字符串加上一个整数。强类型提供更多的安全性。

但是，为了让程序员可以写出极简的代码，很多语言都提供了操作符重载的机制。比如，在 Scala 中，上面的代码是可以被正确执行的（重载了 + 操作符）：

```
scala> "Hello,World"+1
res15: String = Hello,World1

scala> 1+"Hello,World"
res16: String = 1Hello,World
```

但是在 Kotlin 中，由于 Int 类型没有对 + 实现重载，所以情况是这样的：

```
>>> "Hello,World"+1
Hello,World1
>>> 1+"Hello,World"
error: none of the following functions can be called with the arguments supplied:
public final operator fun plus(other: Byte): Int defined in kotlin.Int
public final operator fun plus(other: Double): Double defined in kotlin.Int
public final operator fun plus(other: Float): Float defined in kotlin.Int
public final operator fun plus(other: Int): Int defined in kotlin.Int
public final operator fun plus(other: Long): Long defined in kotlin.Int
```

```
public final operator fun plus(other: Short): Int defined in kotlin.Int
1+"Hello,World"
  ^
```

2. 最优化

静态类型检查可提供有用的信息给编译器，编译器可以使用更有效率的机器指令，实现编译器优化。

3. 抽象化（或模块化）

类型本质上是对较低层次的逻辑单元进行高层次的逻辑抽象。这样我们就可以直接使用类型在较高层次的方式思考，而不是繁重的低层次实现。例如，我们可以将字符串想成一个值，以此取代仅仅是字节的数组。字符串就是一个抽象数据类型。

从 01 到类型，从类型到接口 API，再到软件服务，都可以看做是广义的“类型”范畴。

程序中的变量在程序执行期间，可能会有不同的取值范围，我们可以把变量可取值的最大范围称为这个变量的类型。例如，具有类型 Boolean 的变量 x，在程序执行期间只能取布尔值。指定变量类型的程序设计语言，称为类型化的语言（typed language）。

如果一个语言不限制变量的取值，称为无类型语言（untyped language），我们既可以说它不具有类型，也可以说它具有一个通用类型，这个类型的取值范围是程序中所有可能的值。

类型系统是类型化语言的一个组成部分，它用来计算和跟踪程序中所有表达式的类型，从而判断某段程序是否表现良好（well behaved）。

如果程序语言的语法中含有类型标记，就称该语言是显式类型化的（explicitly typed），否则就称为隐式类型化的（implicitly typed）。

像 C、C++、Java 等语言，都是显式类型化的，而像 ML、Haskell、Groovy 等语言可以省略类型声明，它们的类型系统会自动推断出程序的类型。

4.2 编译时类型与运行时类型

Koltin 是一门强类型的、静态类型、支持隐式类型的显式类型语言。

4.2.1 弱类型与强类型

类型系统最主要的作用是，通过检查类型的运算和转换过程，来减少类型错误的发生。如果一个语言的编译器引入越多的类型检查的限制，就可以称这个语言的类型检查越强，反之越弱。根据类型检查的强弱，我们可将编程语言分为：

- 弱类型语言（Weakly checked language）
- 强类型语言（Strongly checked language）

弱类型语言在运行时会隐式做数据类型转换，强类型语言在运行时会确保不会发生未

经明确转换（显式调用）的类型转换。但是，强和弱只是相对的。

Kotlin 是强类型语言。

4.2.2　静态类型与动态类型

类型检查可发生在编译时期（静态检查）或运行时期（动态检查），这样我们可将编程语言分为：

- 静态类型语言（Statically checked language）
- 动态类型语言（Dynamically Checked language）

静态类型检查是基于编译器来分析源码本身，从而确保类型安全。静态类型检查能让很多 bug 在编码早期被捕捉到，并且它也能优化运行。因为如果编译器在编译时已经证明程序是类型安全的，就不用在运行时进行动态的类型检查，编译过后的代码会更优化，运行更快。

动态类型语言是在运行时期进行类型标记的检查，因为变量所约束的值，可经由运行路径获得不同的标记。关于动态类型有个很形象的说法：

> 当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。
>
> ——詹姆斯·惠特科姆·莱利（James Whitcomb Riley，1849-1916）

Kotlin 是静态类型语言。

4.2.3　显式类型与隐式类型

还有一种区分方法是，根据变量名是否需要显式给出类型的声明，将语言分为：

- 显式类型语言（Explicitly typed language）
- 隐式类型语言（Implicitly typed language）

前者需要在定义变量时显式给出变量的类型，而后者可以使用类型推论来确定变量的类型。

大多数静态类型语言，例如 Java、C/C++ 都是显式类型语言，但是有些则不是，如 Haskell、ML 等，它们可以基于变量的操作来推断其类型；Scala 是静态类型语言，它使用类型推断功能来支持隐式类型。Kotlin 跟 Scala 类似，它也使用类型推断支持隐式类型。但是，在一些场景下也需要显式声明变量的类型，所以我们可以说，Kotlin 同时也是显式类型语言。显式类型语言与隐式类型语言之间的界限很多时候并不是非显即隐的。当然，事物的本身也往往如此。

4.3　根类型 Any

Kotlin 中所有类都有一个共同的超类 Any，如果类声明时没有指定超类，则默认为

Any。我们来看一段代码：

```
>>> val any = Any()
>>> any
java.lang.Object@2e377400
>>> any::class
class kotlin.Any
>>> any::class.java
class java.lang.Object
```

也就是说，Any 在运行时，其类型自动映射成 java.lang.Object。我们知道，在 Java 中 Object 类是所有引用类型的父类。但是不包括基本类型：byte int long 等，基本类型对应的包装类是引用类型，其父类是 Object。而在 Kotlin 中，直接统一，即所有类型都是引用类型，统一继承父类 Any。

Any 是 Java 的等价 Object 类。但是与 Java 不同的是，Kotlin 中语言内部的类型和用户定义类型之间，并没有像 Java 那样划清界限。它们是同一类型层次结构的一部分。

Any 只有 equals()、hashCode() 和 toString() 个方法。其源码是：

```
public open class Any {
    public open operator fun equals(other: Any?): Boolean
    public open fun hashCode(): Int
    public open fun toString(): String
}
```

从 Any 的源码注释中，我们可以看到，判断两个对象是否相等，需要满足以下条件：

- **自反性**：对于任何非空引用值 x，x.equals（x）应返回 true。
- **对称性**：对于任何非空引用值 x 和 y，x.equals（y）应返回 true 当且仅当 y.equals（x）返回 true。
- **传递性**：对于任何非空引用值 x，y，z，如果 x.equals（y）返回 true，y.equals（z）返回 true，那么 x.equals（z）应返回 true
- **一致性**：对于任何非空引用值 x 和 y，多次调用 x.equals（y）始终返回 true 或者始终返回 false。

另外，在 Kotlin 中，操作符 == 会被编译器翻译成调用 equals() 函数。

4.4 基本类型

本节我们来探讨学习 Kotlin 的基础类型（primitive types）：数字、字符、布尔和数组等。



我们知道 Java 的类型分成两种：一种是基本类型，一种是引用类型，它们的本质区别是：基本类型是在堆栈处分配空间存“值”，而引用类型是在堆里面分配空间存

“值”。

Java 的基本类型有：byte、int、short、long、float、double、char、boolean，这些类都有对应的装箱类（引用类型）。

另外，void 也可以算是一种特殊的基本类型，它也有一个装箱类 Void（跟我们后文讲到的 Unit、Nothing 相关）。因为，Void 是不能 new 出来的，也就是不能在堆里面分配空间存对应的值。所以，Void 是一开始在堆栈处分配好空间。所以，将 Void 归成基本类型。

在 Kotlin 中，一切皆是对象。所有类型都是引用类型。没有类似 Java 中的基本类型。但是，可以把 Kotlin 中对应的这几种基本数据类型，理解为 Java 的基本类型的装箱类：

```
Integer.java
public final class Integer extends Number implements Comparable<Integer> {
    /**
     * A constant holding the minimum value an {@code int} can
     * have, -2<sup>31</sup>.
     */
    @Native public static final int   MIN_VALUE = 0x80000000;

    /**
     * A constant holding the maximum value an {@code int} can
     * have, 2<sup>31</sup>-1.
     */
    @Native public static final int   MAX_VALUE = 0x7fffffff;

    /**
     * The {@code Class} instance representing the primitive type
     * {@code int}.
     *
     * @since   JDK1.1
     */
    @SuppressWarnings("unchecked")
    public static final Class<Integer>  TYPE = (Class<Integer>) Class.
        getPrimitiveClass("int");

    ...

}
```

Kotlin 中的 Int 类型：

```
public class Int private constructor() : Number(), Comparable<Int> {
    companion object {
        /**
         * A constant holding the minimum value an instance of Int can have.
         */
```

```
        public const val MIN_VALUE: Int = -2147483648

        /**
         * A constant holding the maximum value an instance of Int can have.
         */
        public const val MAX_VALUE: Int = 2147483647
    }
    ...
}
```

我们通过 Java 的 Integer 封装类，与 Kotlin 的 Int 类的定义相比较，可以看出两者的思想上的同源性。

Kotlin 基本类型的类图结构如图 4-1 所示。

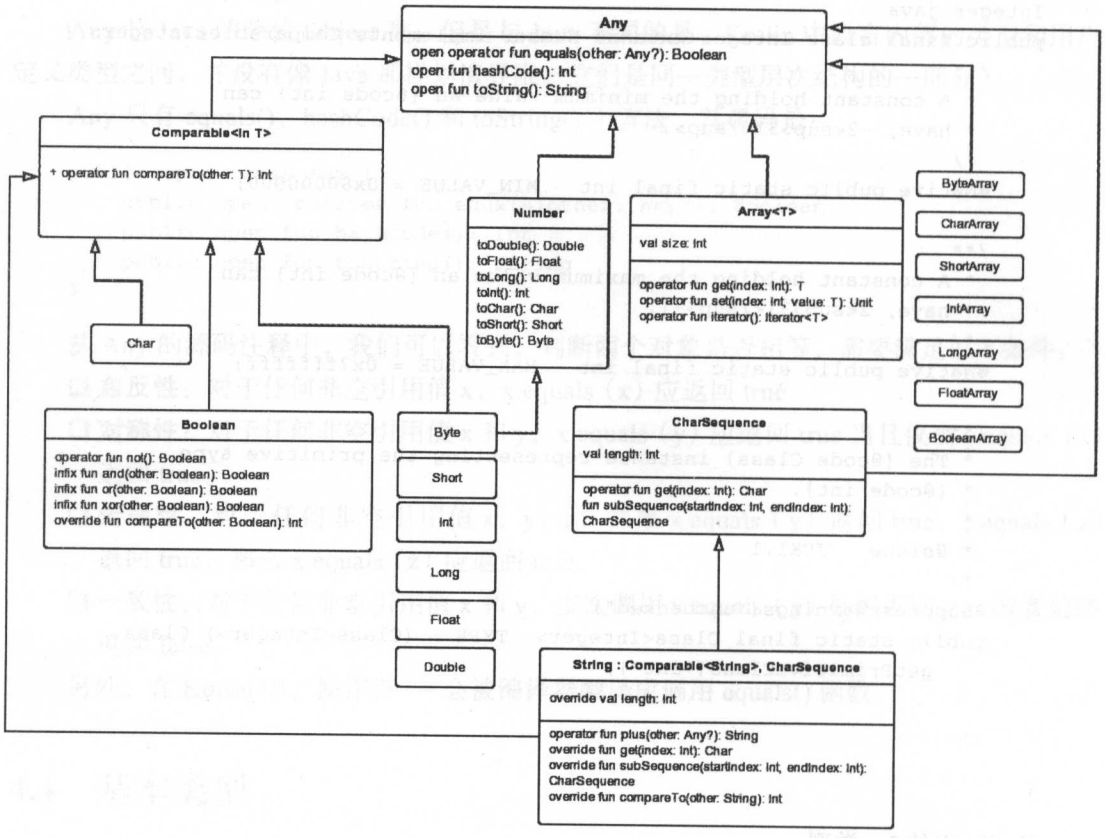


图 4-1 Kotlin 基本类型的类图结构

4.4.1 数字类型

Kotlin 提供了如下的内置类型来表示数字（Number）(与 Java 很相近)：

类型	宽度（Bit）
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

从上节介绍的 Kotlin 基本类型的类图，可以看出这些内置的数据类型，都继承了 Number 和 Comparable 类。例如，Byte 类型的声明：

```
public class Byte private constructor() : Number(), Comparable<Byte> {
    ...
}
```

Kotlin 的数字类型与 Java 基本相同。有一点不同的是，Kotlin 对于数字没有隐式拓宽转换（如 Java 中 int 可以隐式转换为 long）。

注意，在 Kotlin 中字符 Char 不是数字。这些基本数据类型，会在运行时自动优化为 Java 的 double、float、long、int、short、byte。

1. 字面常量值

数值常量字面值（literal constant values）有以下几种：

- 十进制：123
- Long 类型用大写 L 标记：123L
- 十六进制：0x0F
- 二进制：0b00001011

代码示例：

```
>>> 123
123
>>> 123::class
class kotlin.Int
>>> 123::class.java
int
>>> 123L
123
>>> 123L::class
class kotlin.Long
>>> 123L::class.java
long

>>> val b:Byte=128
error: the integer literal does not conform to the expected type Byte
val b:Byte=128
          ^
```

```
>>> val b:Byte=127
>>> b::class
class kotlin.Byte
>>> b::class.java
byte
```

```
>>> 0x0f
15
>>> 0x0F
15
>>> 0b1000
8
```

同样，当我们赋值超过变量的类型取值范围时，编译器会直接抛错。



注意 不支持八进制。

Kotlin 同样支持浮点数的常规表示方法：

- 默认 double：123.5、123.5e10
- Float 用 f 或者 F 标记：123.5f

代码示例如下：

```
>>> 1234.5
1234.5
>>> 1234.5::class
class kotlin.Double
>>> 1234.5::class.java
double
>>> 12.3e10
1.23E11
>>> 12.3e10::class
class kotlin.Double
>>> 456.7f
456.7
>>> 456.7f::class
class kotlin.Float
>>> 456.7f::class.java
float
```

也可以使用数字字面值中的下划线（自 1.1 起），使数字常量更易读：

```
>>> 1_000_000
1000000
>>> 1234_5678_9012_3456L
1234567890123456
```

```
>>> 0xFF_EC_DE_5E
4293713502
>>> 0b11010010_01101001_10010100_10010010
3530134674
```

在 Java 平台数字是物理存储为 JVM 的原生类型，除非我们需要一个可空的引用（如 Int?）或泛型。后者情况下会把数字装箱。

2. 显式转换

由于不同的表示方式，值范围较小类型并不是较大类型的子类型，是不能隐式转换的。代码示例如下：

```
>>> val a: Int? = 1
>>> val b: Long? = a
error: type mismatch: inferred type is Int? but Long? was expected
val b: Long? = a
               ^


>>> val b: Byte = 1
>>> val i: Int = b
error: type mismatch: inferred type is Byte but Int was expected
val i: Int = b
             ^
```

这意味着在不进行显式转换的情况下，不能把 Int 型值赋给一个 Long 变量，也不能把 Byte 型值赋给一个 Int 变量。

我们可以通过显式转换来拓宽数字：

```
>>> val i: Int = b.toInt() //OK: 显式拓宽
```

每个数字类型都继承 Number 抽象类，其中定义了如下的转换函数：

```
toDouble(): Double
toFloat(): Float
toLong(): Long
toInt(): Int
toChar(): Char
toShort(): Short
toByte(): Byte
```

所以，在数字之间的转换，直接调用上面的这些转换函数即可。

3. 运算符 + 重载

缺乏隐式类型转换并不显著，因为类型会从上下文推断出来，而算术运算会有重载做适当转换，例如：

```
val l = 1L + 3 //Long + Int => Long
```

这是通过运算符 + 重载实现的。我们可以在 Long 类的源代码中看到这个 plus 运算符函数的定义：

```
public operator fun plus(other: Byte): Long
public operator fun plus(other: Short): Long
public operator fun plus(other: Int): Long
public operator fun plus(other: Long): Long
public operator fun plus(other: Float): Float
public operator fun plus(other: Double): Double
```

也就是说，编译器会把 1L + 3 翻译成 1L.plus(3)，然后这个传入的参数类型必须是 Byte、Short、Int、Long、Float、Double 中的一种。例如，我们传入一个字符 Char 参数，编译器就会直接抛错：

```
>>> 'a'
a
>>> 'a'::class
class kotlin.Char
>>> 'a'::class.java
char
>>> 1L+'a'
error: none of the following functions can be called with the arguments supplied:
public final operator fun plus(other: Byte): Long defined in kotlin.Long
public final operator fun plus(other: Double): Double defined in kotlin.Long
public final operator fun plus(other: Float): Float defined in kotlin.Long
public final operator fun plus(other: Int): Long defined in kotlin.Long
public final operator fun plus(other: Long): Long defined in kotlin.Long
public final operator fun plus(other: Short): Long defined in kotlin.Long
1L+'a'
  ^
```

4. 运算

Kotlin 支持数字运算的标准集，运算被定义为相应的类成员（但编译器会将函数调用优化为相应的指令）。

对于位运算，没有特殊字符来表示，而只可用中缀方式调用命名函数（infix fun），例如：

```
val x = (1 shl 2) and 0x000FF000
```

这是完整的位运算列表（只用于 Int 和 Long）：

- shl(bits)——有符号左移（Java 的 <<）
- shr(bits)——有符号右移（Java 的 >>）
- ushr(bits)——无符号右移（Java 的 >>>）
- and(bits)——位与
- or(bits)——位或
- xor(bits)——位异或
- inv()——位非

4.4.2　字符类型与转义符

字符（Character）用 Char 类型表示。它们不能直接当作数字：

```
fun check(c: Char) {
    if (c == 1) { // 错误：类型不兼容
        //……
    }
}
```

字符字面值用单引号括起来：'1'。

特殊字符可以用反斜杠转义。Kotlin 支持如下转义字符（Escape character）：

```
\t
\b
\n
\r
\'
\"
\\
\$
```

编码其他字符要用 Unicode 转义序列语法，例如：'\uFF00'。

Char 类的函数接口定义如下：

```
public class Char private constructor() : Comparable<Char> {
    public override fun compareTo(other: Char): Int
    public operator fun plus(other: Int): Char
    public operator fun minus(other: Char): Int
    public operator fun minus(other: Int): Char
    public operator fun inc(): Char
    public operator fun dec(): Char
    public operator fun rangeTo(other: Char): CharRange
    public fun toByte(): Byte
    public fun toChar(): Char
    public fun toShort(): Short
    public fun toInt(): Int
    public fun toLong(): Long
    public fun toFloat(): Float
    public fun toDouble(): Double


}
```

下面用代码示例这些函数的使用：

如果两个字符相等：

```
>>> 'a'.compareTo('a')
0
```

如果两个字符不相等：

```
>>> 'a'.compareTo('b')
-1
>>> 'a'.compareTo('c')
-1
>>> 'b'.compareTo('a')
1
>>> 'c'.compareTo('a')
1
```

Char 字符只重载了加上 Int 类型的数字的 + 运算符：

```
>>> 'a'+1
b

>>> 'a'+1L
error: the integer literal does not conform to the expected type Int
'a'+1L
```

所以，当我们把一个 Char 类型值和不是 Int 类型的值相加，就报错了。

相减：

```
>>> 'a'-1
`
>>> 'c'-'a'
2
```

自增计算：

```
>>> var a='a'
>>> val b=a++
>>> a
b
>>> b
a
>>> val c=++a
>>> c
c
```

我们不能在字符的字面量上直接使用 ++：

```
>>> 'a'++
error: variable expected
'a'++
^

>>> ++'a'
error: variable expected
++'a'
  ^
```

范围：

```
>>> 'a'.rangeTo('z')
a..z
>>> for(c in 'a'..'z') {print(c)}

abcdefghijklmnopqrstuvwxyz
```

Char 的显式类型转换函数如下：

```
/** Returns the value of this character as a `Byte`. */
public fun toByte(): Byte
/** Returns the value of this character as a `Char`. */
public fun toChar(): Char
/** Returns the value of this character as a `Short`. */
public fun toShort(): Short
/** Returns the value of this character as a `Int`. */
public fun toInt(): Int
/** Returns the value of this character as a `Long`. */
public fun toLong(): Long
/** Returns the value of this character as a `Float`. */
public fun toFloat(): Float
/** Returns the value of this character as a `Double`. */
public fun toDouble(): Double
```

例如，我们显式地把字符转换为 Int 数字：

```
fun decimalDigitValue(c: Char): Int {
    if (c !in '0'..'9')
        throw IllegalArgumentException("Out of range")
    return c.toInt() - '0'.toInt() //显式转换为数字
}
```

测试代码：

```
>>> decimalDigitValue('a')
java.lang.IllegalArgumentException: Out of range
    at Line24.decimalDigitValue(Unknown Source)

>>> decimalDigitValue('1')
1
```

4.4.3　布尔类型

Kotlin 的布尔类型用 Boolean 类来表示，有两个值：true 和 false，示例代码如下：

```
>>> true::class
class kotlin.Boolean
>>> true::class.java
boolean
```

对应 Java 中的 boolean 类型。

Boolean 类型的运算符函数如下：

```
public operator fun not(): Boolean
public infix fun and(other: Boolean): Boolean
public infix fun or(other: Boolean): Boolean
public infix fun xor(other: Boolean): Boolean
public override fun compareTo(other: Boolean): Int
```

分别对应：

- ! 逻辑非：not()
- && 短路逻辑与：and()
- || 短路逻辑或：or()
- 异或（相同 false，不同 true）：xor

另外，Boolean 还继承实现了 Comparable 的 compareTo() 函数。

代码示例如下：

```
>>> !true
false
>>> true.not()
false
>>> true && true
true
>>> true.and(false)
false
>>> true || false
true
>>> false.or(false)
false
>>> true xor true
false
>>> true xor false
true
>>> false xor false
false
>>> true > false
true
>>> true < false
false
>>> true.compareTo(false)
1
>>> true.compareTo(false)
1
>>> true.compareTo(true)
0
>>> false.compareTo(true)
-1
```

4.4.4 字符串类型

Kotlin 的字符串用 String 类型表示，对应 Java 中的 java.lang.String。字符串是不可变

的，代码示例如下：

```
>>> "abc"::class
class kotlin.String
>>> "abc"::class.java
class java.lang.String
```

另外，在 Kotlin 中，String 同样是 final 不可继承的，代码示例如下：

```
>>> class MyString:String
error: this type is final, so it cannot be inherited from
class MyString:String
               ^
```

1. 索引运算符 s[i]

字符串的元素——字符可以使用索引运算符 s[i] 来访问：

```
>>> val s="abc"
>>> s
abc
>>> s[0]
a
```

当我们下标越界时，会抛越界错误：

```
>>> s[-1]
java.lang.StringIndexOutOfBoundsException: String index out of range: -1
    at java.lang.String.charAt(String.java:646)

>>> s[3]
java.lang.StringIndexOutOfBoundsException: String index out of range: 3
    at java.lang.String.charAt(String.java:646)
```

从出错信息我们可以看出，索引运算符 s[i] 会被翻译成 java.lang.String.charAt()，背后调用的是 Java 的 String 类，其调用的方法是：

```
public char charAt(int index) {
    if ((index < 0) || (index >= value.length)) {
        throw new StringIndexOutOfBoundsException(index);
    }
    return value[index];
}
```

2. for 循环迭代字符串

我们可以用 for 循环迭代字符串：

```
>>> for(c in "abc") { println(c)  }
a
b
c
```

关于字符串 String 类的完整操作方法，我们可以看下源码：

```
public class String : Comparable<String>, CharSequence {
    companion object {}

    /**
     * Returns a string obtained by concatenating this string with the string
         representation of the given [other] object.
     */
    public operator fun plus(other: Any?): String

    public override val length: Int

    public override fun get(index: Int): Char

    public override fun subSequence(startIndex: Int, endIndex: Int): CharSequence

    public override fun compareTo(other: String): Int
}
```

类似的，字符串有一个 length 属性：

```
>>> "abc".length
3
```

3. 重载 + 操作符

字符串类重载了 + 操作符，作用对象可以是任何对象，包括空引用：

```
>>> "abc".plus(true)
abctrue
>>> "abc"+false
abcfalse
>>> "abc"+1
abc1
>>> "abc"+1.20
abc1.2
>>> "abc"+100L
abc100
>>> "abc"+"cdef"
abccdef
>>> "abc"+null
abcnull
>>> "abc"+'z'
abcz
>>> "abc"+arrayOf(1,2,3,4,5)
abc[Ljava.lang.Integer;@3d6f0054
```

截取字符串的子串：

```
>>> "abc".subSequence(0,1)
a
```

```
>>> "abc".subSequence(0,2)
ab
>>> "abc".subSequence(0,3)
abc
>>> "abc".subSequence(0,4)
java.lang.StringIndexOutOfBoundsException: String index out of range: 4
    at java.lang.String.substring(String.java:1951)
    at java.lang.String.subSequence(String.java:1991)
```

4. 字符串字面值

字符串的字面值可以包含原生字符串，可以包含换行符和任意文本，也可以是带有转义字符（escape charactor）的转义字符串，示例代码如下：

```
>>> val s = "Hello,World!\n\n\n"
>>> s
Hello,World!



>>>
```

转义采用传统的反斜杠方式。

原生字符串使用三个引号（"""）分界符括起来，内部没有转义并且可以包含换行和任何其他字符：

```
>>> val text = """
...     for (c in "abc")
...         print(c)
... """
>>> text

    for (c in "foo")
        print(c)

>>>
```

另外，在 package kotlin.text 下面的 Indent.kt 代码中，Kotlin 还定义了 String 类的扩展函数：

```
fun String.trimMargin(marginPrefix: String = "|"): String
fun String.trimIndent(): String
```

我们可以使用 trimMargin()、trimIndent() 裁剪函数来去除前导空格。可以看出，trimMargin() 函数默认使用 "|" 来作为边界字符：

```
>>> val text = """
... |理论是你知道是这样，但它却不好用。
... |实践是它很好用，但你不知道是为什么。
... |程序员将理论和实践结合到一起：
```

```
... | 既不好用，也不知道是为什么。
...     """
>>> text.trimMargin()
理论是你知道是这样，但它却不好用。
实践是它很好用，但你不知道是为什么。
程序员将理论和实践结合到一起：
既不好用，也不知道是为什么。
```

我们可以看出，text 字符串被裁剪了。

默认使用|作为边界前缀，但你可以选择其他字符并作为参数传入，比如 trimMargin(">")。

还有一个 trimIndent() 函数，则是把字符串行的左边空白对齐切割：

```
>>> val text="""
...                 Hello
...                       World!
... """
>>> text.trimIndent()
Hello
    World!
>>> val text="""
...               Hello,
...         World!
... """
>>> text.trimIndent()
 Hello,
World!
```

5. 字符串模板

字符串可以包含模板表达式，即一些小段代码，会求值并把结果合并到字符串中。

模板表达式以美元符（$）开头，由一个简单的名字构成，示例代码如下：

```
>>> val h=100
>>> val str = "A hundred is $h"
>>> str
A hundred is 100
```

或者用花括号扩起来的任意表达式：

```
>>> val s = "abc"
>>> val str = "$s.length is ${s.length}"
>>> str
abc.length is 3
```

原生字符串和转义字符串内部都支持模板：

```
>>> val price=9.9
>>> val str="""Price is $$price"""
>>> str
```

```
Price is $9.9
>>> val str="Price is $$price"
>>> str
Price is $9.9
>>> val quantity=100
>>> val str="Quantity is $quantity"
>>> str
Quantity is 100
>>> val str="""Quantity is $quantity"""
>>> str
Quantity is 100
```

4.4.5　数组类型

数组在 Kotlin 中使用 Array 类来表示，它定义了 get 和 set 函数（映射到重载运算符 []）和 size 属性，以及一个用于变量数组的 iterator() 函数：

```
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
    operator fun set(index: Int, value: T): Unit
    operator fun iterator(): Iterator<T>
    //……
}
```

可以使用函数 arrayOf() 来创建一个数组并传递元素值给它。这个函数签名如下：

```
public inline fun <reified @PureReifiable T> arrayOf(vararg elements: T): Array<T>
```

其中，vararg 表示是一个参数个数是一个变量。

例如，arrayOf(1, 2, 3) 创建了 array [1, 2, 3]：

```
>>> arrayOf(1,2,3)
[Ljava.lang.Integer;@4a37191a
>>> arrayOf(1,2,3)::class
class kotlin.Array
>>> arrayOf(1,2,3)::class.java
class [Ljava.lang.Integer;
```

另外，Kotlin 还允许不同类型元素放到一个数组中，例如：

```
>>> val arr = arrayOf(1,"2",true)
>>> arr
[Ljava.lang.Object;@61af1510
>>> arr.forEach{ println(it) }
1
2
true
>>> arr.forEach{ println(it::class) }
```

```
class kotlin.Int
class kotlin.String
class kotlin.Boolean
```

Kotlin 自动把这个数组元素的类型升级为 java.lang.Object，同时，由于 Kotlin 拥有类型推断的功能，我们仍然可以看到每个数组元素对应的类型。

函数 arrayOfNulls() 可以用于创建一个指定大小、元素都为空的数组。这个特殊的空数组在创建时，需要指定元素的类型。如果不指定，直接按照下面这样写，会报错：

```
>>> arrayOfNulls(10)
error: type inference failed: Not enough information to infer parameter T in
    fun <reified T> arrayOfNulls(size: Int): Array<T?>
Please specify it explicitly.

arrayOfNulls(10)
^
```

也就是说，我们要指定元素的类型：

```
>>> arrayOfNulls<Int>(10)
[Ljava.lang.Integer;@77c10a5f
>>> arrayOfNulls<Int>(10).forEach{println(it)}
null
null
null
null
null
null
null
null
null
null
```

数组 Array 类还提供了一个构造函数：

```
public inline constructor(size: Int, init: (Int) -> T)
```

第 1 个参数是数组大小，第 2 个参数是一个初始化函数类型的参数（关于函数类型，我们将在后面章节介绍）。

代码示例如下：

```
>>> val square = Array(10, { i -> (i*i)})
>>> square
[Ljava.lang.Integer;@6f9e08d4
>>> square.forEach{ println(it) }
0
1
4
9
```

```
16
25
36
49
64
81
```

如上所述，[] 运算符代表调用成员函数 get() 和 set()。

代码示例如下：

```
>>> square[3]
9

>>> square[3]=1000
>>> square.forEach{ println(it) }
0
1
4
1000
16
25
36
49
64
81
```

与 Java 不同的是，Kotlin 中数组不是型变的（invariant）。Kotlin 中，我们不能把 Array<String> 赋值给 Array<Any>。这地方 Kotlin 类型检查的限制强于 Java 的数组类型。

代码示例如下：

```
>>> val arrstr = arrayOf<String>("1","2","3")
>>> arrstr
[Ljava.lang.String;@39374689
>>> var arrany = arrayOf<Any>(Any(),Any(),Any())
>>> arrany
[Ljava.lang.Object;@156324b
>>> arrany = arrstr
error: type mismatch: inferred type is Array<String> but Array<Any> was expected
arrany = arrstr
         ^
```

原生数组类型

Kotlin 也有无装箱开销的专门的类来表示原生类型数组，这些原生数组类如下所示：

```
BooleanArray
ByteArray
CharArray
ShortArray
IntArray
```

```
LongArray
FloatArray
DoubleArray
BooleanArray
```

这些类和 Array 并没有继承关系，但它们有同样的函数和属性集，也都有相应的工厂方法：

```
public fun doubleArrayOf(vararg elements: Double): DoubleArray
public fun floatArrayOf(vararg elements: Float): FloatArray
public fun longArrayOf(vararg elements: Long): LongArray
public fun intArrayOf(vararg elements: Int): IntArray
public fun charArrayOf(vararg elements: Char): CharArray
public fun shortArrayOf(vararg elements: Short): ShortArray
public fun byteArrayOf(vararg elements: Byte): ByteArray
public fun booleanArrayOf(vararg elements: Boolean): BooleanArray
```

代码示例：

```
>>> val x: IntArray = intArrayOf(1, 2, 3)
>>> x[0]
1
```

4.5 可空类型

在 3.9 节中，我们已经介绍了 Kotlin 的空指针安全的相关内容，本节我们从类型的角度再来认识一下 Kotlin 中的可空类型。

4.5.1 Kotlin 中的 null

可空类型（Nullable Types）Any? 是 Kotlin 类型系统的一个特性，主要是为了解决 Java 中的令人头疼的 NullPointerException 问题。Kotlin 把可空性（nullability）作为类型系统的一部分，Kotlin 编译器可以直接在编译过程中发现许多可能的错误，并减少在运行时抛出异常的可能性。Kotlin 的类型系统和 Java 相比，首要的区别就是 Kotlin 对可空类型的显式支持。

在 Kotlin 中，针对 Java 中的 null 的杂乱局面进行了整顿，作了清晰的界定，并在编译器级别强制规范了可空 null 变量类型的使用。

我们来看一下 Kotlin 中关于 null 的一些有趣的运算。

null 跟 null 是相等的：

```
>>> null==null
true
>>> null!=null
false
```

null 这个值比较特殊，null 不是 Any 类型：

```
>>> null is Any
false
```

但是，null 是 Any? 类型：

```
>>> null is Any?
true
```

我们来看看 null 对应的类型到底是什么：

```
>>> var a=null
>>> a
null
>>> a=1
error: the integer literal does not conform to the expected type Nothing?
a=1
  ^
```

从报错信息可以看出，null 的类型是 Nothing?。关于 Nothing? 我们将在下一小节中介绍。

我们可以对 null 进行加法运算：

```
>>> "1"+null
1null

>>> null+20
null20
```

对应的重载运算符的函数定义在 kotlin/Library.kt 里面：

```
package kotlin

import kotlin.internal.PureReifiable

public fun Any?.toString(): String
public operator fun String?.plus(other: Any?): String
```

但是，反过来就不行了：

```
>>> 1+null
error: none of the following functions can be called with the arguments supplied:
public final operator fun plus(other: Byte): Int defined in kotlin.Int
public final operator fun plus(other: Double): Double defined in kotlin.Int
public final operator fun plus(other: Float): Float defined in kotlin.Int
public final operator fun plus(other: Int): Int defined in kotlin.Int
public final operator fun plus(other: Long): Long defined in kotlin.Int
public final operator fun plus(other: Short): Int defined in kotlin.Int
1+null
 ^
```

这是因为 Int 没有重载传入 null 参数的 plus() 函数。

4.5.2 可空类型 String? 与安全调用 ?.

我们来看一个例子，下面是计算字符串长度的简单 Java 方法：

```
public static int getLength1(String str) {
        return str.length();
    }
```

我们已经习惯了在这样的 Java 代码中，加上这样的空判断处理：

```
public static int getLength2(String str) throws Exception {
    if (null == str) {
        throw new Exception("str is null");
    }

    return str.length();
}
```

而在 Kotlin 中，当同样写一个可能为 null 参数的函数时：

```
fun getLength1(str: String): Int {
        return str.length
    }
```

当传入一个 null 参数时：

```
@Test fun testGetLength1() {
    val StringUtilKt = StringUtilKt()
    StringUtilKt.getLength1(null)
}
```

编译器就直接编译失败：

```
e: /Users/jack/easykotlin/chapter4_type_system/src/test/kotlin/com/easy/kotlin/
    StringUtilKtTest.kt: (15, 33): Null can not be a value of a non-null type String
:compileTestKotlin FAILED
FAILURE: Build failed with an exception.
* What went wrong:
Execution failed for task ':compileTestKotlin'.
> Compilation error. See log for more details
```

如果我们使用 IDEA，会在编码时就直接提示错误了，如图 4-2 所示。

Kotlin 就是这样通过编译时强制排除空指针的错误，大大减少了出现 NPE 的可能。

另外，如果我们确实需要传入一个可空的参数，可以使用可空类型 String? 来声明一个可以指向空指针的变量。

可空类型可以用来标记任何一个变量，来表明这个变量是可空的（Nullable）。例如：Char?、Int?、MineType?（自定义的类型）等等。

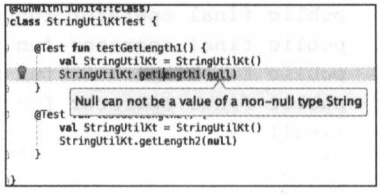


图 4-2 IDEA 会在编码时直接提示错误

用示例代码来更加简洁地说明：

```
>>> var x:String="x"
>>> x=null
error: null can not be a value of a non-null type String
x=null
  ^

>>> var y:String?="y"
>>> y=null
>>> y
null
```

可以看出：普通 String 类型是不允许指向 null 的，而可空 String? 类型可以指向 null。

下面我们来尝试使用一个可空变量来调用函数：

```
>>> fun getLength2(str: String?): Int? = str.length
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable
    receiver of type String?
fun getLength2(str: String?): Int? = str.length
                                     ^
```

编译器直接报错，告诉我们，变量 str: String? 是可空的类型，调用只能通过安全调用 (?.) 或者非空断言调用 (!!.)。

另外，如果不需要捕获异常来处理，我们可以使用 Kotlin 里面的安全调用符 (?.)，如图 4-3 所示。

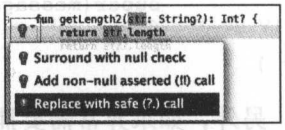


图 4-3　Kotlin 里面的安全调用符

代码示例：

```
fun getLength2(str: String?): Int? {
    return str?.length
}
```

测试代码：

```
@Test fun testGetLength2() {
    val StringUtilKt = StringUtilKt()
    println(StringUtilKt.getLength2(null)) //null
    Assert.assertTrue(3 == StringUtilKt.getLength2("abc"))
}
```

我们可以看出，当使用安全调用 (?.)，代码安静地执行，输出了 null。

如果我们确实想写一个出现空指针异常的代码，那就使用可能出现空指针的断言调用符 (!!.)。

代码示例：

```
fun getLength3(str: String?): Int? {
    return str!!.length
}
```

测试代码：

```
@Test fun testGetLength3() {
    val StringUtilKt = StringUtilKt()
    println(StringUtilKt.getLength3(null))
    Assert.assertTrue(3 == StringUtilKt.getLength3("abc"))
}
```

上面的代码就与 Java 里面差不多了，运行会直接抛出空指针异常：

```
kotlin.KotlinNullPointerException
    at com.easy.kotlin.StringUtilKt.getLength3(StringUtilKt.kt:16)
    at com.easy.kotlin.StringUtilKtTest.testGetLength3(StringUtilKtTest.kt:28)
```

这里的 KotlinNullPointerException 是 KotlinNullPointerException.java 代码，继承了 Java 中的 java.lang.NullPointerException，它的源代码如下：

```
package kotlin;

public class KotlinNullPointerException extends NullPointerException {
    public KotlinNullPointerException() {
    }

    public KotlinNullPointerException(String message) {
        super(message);
    }
}
```

另外，如果异常需要捕获到进行特殊处理的场景，在 Kotlin 中仍然使用 try...catch 捕获并处理异常。

4.5.3 可空性的实现原理

我们来看一段 Kotlin 的可空类型的示例代码如下：

```
fun testNullable1(x: String, y: String?): Int {
    return x.length
}

fun testNullable2(x: String, y: String?): Int? {
    return y?.length
}

fun testNullable3(x: String, y: String?): Int? {
    return y!!.length
}
```

我们使用 IDEA 的 Kotlin 插件来看一下可空类型的安全调用的等价 Java 代码。

打开 IDEA 的 Tools → Kotlin → Show Kotlin Bytecode，如图 4-4 所示。

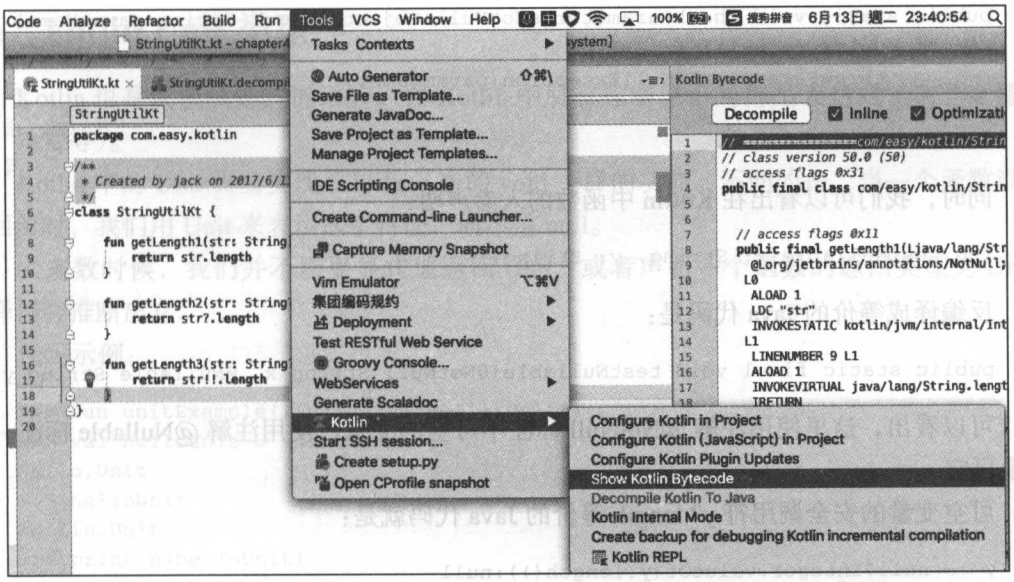


图 4-4　打开 IDEA 的 Tools 菜单

然后，点击 Decompile，我们可以得到反编译的 Java 代码：

```
public final class NullableTypesKt {
    public static final int testNullable1(@NotNull String x, @Nullable String y) {
        Intrinsics.checkParameterIsNotNull(x, "x");
        return x.length();
    }

    @Nullable
    public static final Integer testNullable2(@NotNull String x, @Nullable String y) {
        Intrinsics.checkParameterIsNotNull(x, "x");
        return y != null?Integer.valueOf(y.length()):null;
    }

    @Nullable
    public static final Integer testNullable3(@NotNull String x, @Nullable String y) {
        Intrinsics.checkParameterIsNotNull(x, "x");
        if(y == null) {
            Intrinsics.throwNpe();
        }

        return Integer.valueOf(y.length());
    }
}
```

在不可空变量调用函数之前，都检查了是否为空，使用的是 kotlin.jvm.internal.Intrinsics 这个 Java 类里面的 checkParameterIsNotNull 方法。如果是 null 就抛出异常：

```
public static void checkParameterIsNotNull(Object value, String paramName) {
    if (value == null) {
        throwParameterIsNullException(paramName);
    }
}
```

同时，我们可以看出在 Kotlin 中函数的入参声明：

```
fun testNullable(x: String, y: String?)
```

反编译成等价的 Java 代码是：

```
public static final void testNullable(@NotNull String x, @Nullable String y)
```

可以看出，这里使用注解 @NotNull 标注不可空的变量，使用注解 @Nullable 标注一个变量可空。

可空变量的安全调用符 y?.length 等价的 Java 代码就是：

```
y != null?Integer.valueOf(y.length()):null
```

可空变量的断言调用 y!!.length 等价的 Java 代码是：

```
if(y == null) {
    Intrinsics.throwNpe();
}
return Integer.valueOf(y.length());
```

4.5.4 可空类型层次体系

就像 Any 是在非空类型层次结构的根，Any? 是可空类型层次的根。由于 Any? 是 Any 的超集，所以，Any? 是 Kotlin 的类型层次结构的最顶端，如图 4-5 所示。

代码示例如下：

```
>>> 1 is Any
true

>>> 1 is Any?
true

>>> null is Any
false

>>> null is Any?
true


>>> Any() is Any?
true
```

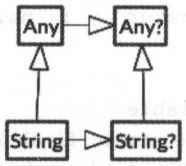


图 4-5 Any? 是 Kotlin 的类型层次结构的最顶端

4.6　kotlin.Unit 类型

Kotlin 也是面向表达式的语言。在 Kotlin 中所有控制流语句都是表达式（除了变量赋值、异常等）。

Kotlin 中的 Unit 类型实现了与 Java 中的 void 一样的功能。不同的是，当一个函数没有返回值时，我们用 Unit 来表示这个特征，而不是 null。

大多数时候，我们并不需要显式地返回 Unit，或者声明一个函数的返回类型为 Unit。编译器会推断出它。

代码示例：

```
>>> fun unitExample(){println("Hello,Unit")}
>>> val helloUnit = unitExample()
Hello,Unit
>>> helloUnit
kotlin.Unit
>>> println(helloUnit)
kotlin.Unit
```

下面几种写法是等价的：

```
@RunWith(JUnit4::class)
class UnitDemoTest {
    @Test fun testUnitDemo() {
        val ur1 = unitReturn1()
        println(ur1) //kotlin.Unit
        val ur2 = unitReturn2()
        println(ur2) //kotlin.Unit
        val ur3 = unitReturn3()
        println(ur3) //kotlin.Unit
    }

    fun unitReturn1() {          //什么都不写

    }

    fun unitReturn2() {          //显示地返回 Unit
        return Unit
    }

    fun unitReturn3(): Unit {    //函数声明的时候返回 Unit 类型
    }
}
```

总的来说，这个 Unit 类型并没有什么特别之处，它的源码是：

```
package kotlin

/**
```

```
 * The type with only one value: the Unit object. This type corresponds to the
     `void` type in Java.
 */
public object Unit {
    override fun toString() = "kotlin.Unit"
}
```

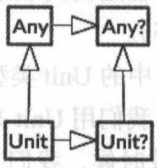


图 4-6 可空的 Unit? 的父类型是 Any?

与任何其他类型一样，Unit 类型的父类型是 Any。如果是一个可空的 Unit?，它的父类型是 Any?，如图 4-6 所示。

4.7 kotlin.Nothing 类型

Kotlin 中没有 Java 和 C 中的相类似函数没有返回值的标记 void，但是拥有一个对应 Nothing。在 Java 中，返回 void 的方法，其返回值 void 是无法被访问到的：

```
public class VoidDemo {
    public void voidDemo() {
        System.out.println("Hello,Void");
    }
}
```

测试代码：

```
@org.junit.runner.RunWith(org.junit.runners.JUnit4.class)
public class VoidDemoTest {
    @org.junit.Test
    public void testVoid() {
        VoidDemo voidDemo = new VoidDemo();
        void v = voidDemo.voidDemo(); //没有 void 变量类型，无法访问到 void 返回值
        System.out.println(voidDemo.voidDemo());      //error: 'void' type not
                                                      //allowed here
    }
}
```

在 Java 中，void 不能是变量的类型。也不能被当做值打印输出。但是，在 Java 中有个包装类 Void 是 void 的自动装箱类型。如果你想让一个方法返回类型永远是 null 的话，可以把返回类型置为这个大写的 V 的 Void 类型。

代码示例：

```
public Void voidDemo() {
    System.out.println("Hello,Void");
    return null;
}
```

测试代码：

```
@org.junit.runner.RunWith(org.junit.runners.JUnit4.class)
public class VoidDemoTest {
    @org.junit.Test
    public void testVoid() {
        VoidDemo voidDemo = new VoidDemo();
        Void v = voidDemo.voidDemo(); //Hello,Void
        System.out.println(v); //null
    }
}
```

这个 Void 就是 Kotlin 中的 Nothing?。它的唯一可被访问到的返回值也是 null。

在 Kotlin 类型层次结构的最底层就是类型 Nothing，如图 4-7 所示。

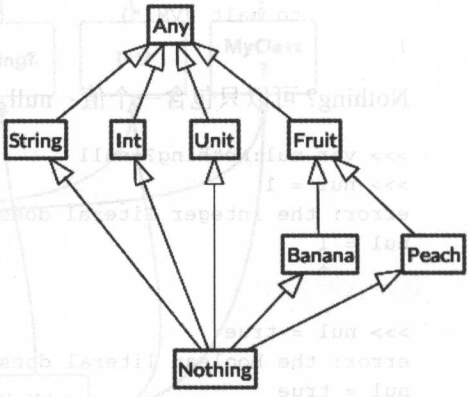


图 4-7　最底层的类型是 Nothing

正如它的名字 Nothing 所暗示的，Nothing 是没有实例的类型。

代码示例如下：

```
>>> Nothing() is Any
error: cannot access '<init>': it is private in 'Nothing'
Nothing() is Any
^
```



注意　Unit 与 Nothing 之间的区别：Unit 类型表达式计算结果的返回类型是 Unit。Nothing 类型的表达式计算结果是永远不会返回的（跟 Java 中的 void 相同）。

例如，throw 关键字中断的表达式的计算，并抛出堆栈的功能。所以，一个 throw Exception 的代码就是返回 Nothing 的表达式。代码示例：

```
fun formatCell(value: Double): String =
    if (value.isNaN())
        throw IllegalArgumentException("$value is not a number")  //Nothing
    else
        value.toString()
```

再例如，Kotlin 的标准库里面的 exitProcess 函数：

```
@file:kotlin.jvm.JvmName("ProcessKt")
@file:kotlin.jvm.JvmVersion
package kotlin.system

/**
 * Terminates the currently running Java Virtual Machine. The
 * argument serves as a status code; by convention, a nonzero status
 * code indicates abnormal termination.
 *
```

```
 * This method never returns normally.
 */
@kotlin.internal.InlineOnly
public inline fun exitProcess(status: Int): Nothing {
    System.exit(status)
    throw RuntimeException("System.exit returned normally, while it was supposed
        to halt JVM.")
}
```

Nothing? 可以只包含一个值：null。代码示例：

```
>>> var nul:Nothing?=null
>>> nul = 1
error: the integer literal does not conform to the expected type Nothing?
nul = 1
      ^

>>> nul = true
error: the boolean literal does not conform to the expected type Nothing?
nul = true
      ^

>>> nul = null
>>> nul
null
```

从上面的代码示例中，我们可以看出：Nothing? 唯一允许的值是 null，被用作任何可空类型的空引用，如图 4-8 所示。

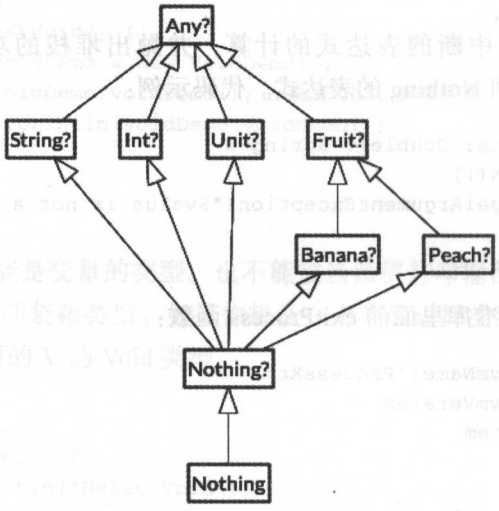


图 4-8 Nothing? 唯一允许的值是 null

综上所述，我们可以看出 Kotlin 有一个简单而一致的类型系统。Any? 是整个类型体系

的顶部，Nothing 是底部，如图 4-9 所示。

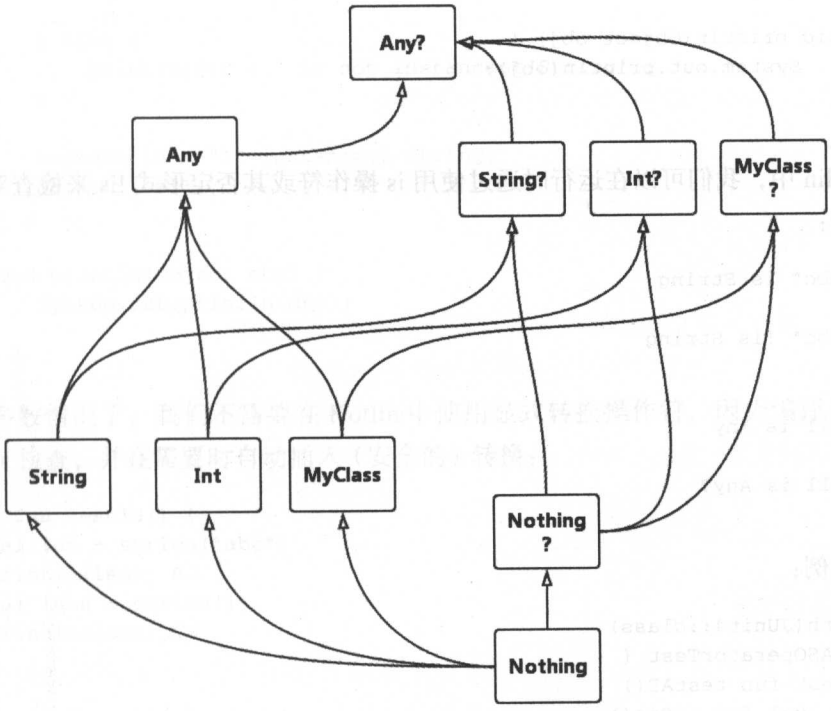


图 4-9　Kotlin 有一个简单而一致的类型系统

4.8　类型检测与类型转换

4.8.1　is 与 !is 运算符

is 运算符可以检查对象是否与特定的类型兼容（“兼容”的意思是：此对象是该类型或者派生于该类型）。

is 运算符用来检查对象（变量）是否属于某数据类型（如 Int、String、Boolean 等）。C# 里面也有这个运算符。

is 运算符类似于 Java 的 instanceof：

```
@org.junit.runner.RunWith(org.junit.runners.JUnit4.class)
public class TypeSystemDemo {
    @org.junit.Test
    public void testVoid() {
        if ("abc" instanceof String) {
            println("abc is instanceof String");
        } else {
            println("abc is not instanceof String");
```

```
        }
    }

    void println(Object obj) {
        System.out.println(obj);
    }
}
```

在 Kotlin 中，我们可以在运行时通过使用 is 操作符或其否定形式 !is 来检查对象是否符合给定类型：

```
>>> "abc" is String
true
>>> "abc" !is String
false

>>> null is Any
false
>>> null is Any?
true
```

代码示例：

```
@RunWith(JUnit4::class)
class ASOperatorTest {
    @Test fun testAS() {
        val foo = Foo()
        val goo = Goo()
        println(foo is Foo) //true 自己
        println(goo is Foo) //子类 is 父类 = true
        println(foo is Goo) //父类 is 子类 = false
        println(goo is Goo) //true 自己
    }
}

open class Foo
class Goo : Foo()
```

类型自动转换

在 Java 代码中，当我们使用 str instanceof String 来判断其值为 true 的时候，要想使用 str 变量，还需要显式的强制转换类型：

```
@org.junit.runner.RunWith(org.junit.runners.JUnit4.class)
public class TypeSystemDemo {
    @org.junit.Test
    public void testVoid() {
        Object str = "abc";
        if (str instanceof String) {
            int len = ((String)str).length();  //显式的强制转换类型为 String
```

```
            println(str + " is instanceof String");
            println("Length: " + len);

        } else {
            println(str + " is not instanceof String");
        }

        boolean is = "1" instanceof String;
        println(is);
    }

    void println(Object obj) {
        System.out.println(obj);
    }
}
```

而大多数情况下，我们不需要在 Kotlin 中使用显式转换操作符，因为编译器会跟踪不可变值的 is 检查，并在需要时自动插入（安全的）转换：

```
@Test fun testIS() {
    val len = strlen("abc")
    println(len)  // 3
    val lens = strlen(1)
    println(lens) // 1
}

fun strlen(ani: Any): Int {
    if (ani is String) {
        return ani.length
    } else if (ani is Number) {
        return ani.toString().length
    } else if (ani is Char) {
        return 1
    } else if (ani is Boolean) {
        return 1
    }


    print("Not A String")
    return -1
}
```

4.8.2　as 运算符

as 运算符用于执行引用类型的显式类型转换。如果要转换的类型与指定的类型兼容，转换就会成功进行；如果类型不兼容，使用 as? 运算符就会返回值 null。

代码示例：

```
>>> open class Foo
>>> class Goo:Foo()
```

```
>>> val foo = Foo()
>>> val goo = Goo()


>>> foo as Goo
java.lang.ClassCastException: Line69$Foo cannot be cast to Line71$Goo

>>> foo as? Goo
null

>>> goo as Foo
Line71$Goo@73dce0e6
```

我们可以看出，在 Kotlin 中，子类是禁止转换为父类型的。

按照 Liskov 替换原则，父类转换为子类是对 OOP 的严重违反，不提倡、也不建议。严格来说，父类是不能转换为子类的，子类包含了父类所有的方法和属性，而父类则未必具有和子类同样成员范围，所以这种转换是不允许的，即便是两个具有父子关系的空类型，也是如此。

4.9 本章小结

在本章中，我们停下脚步，仔细深入地去探讨了 Kotlin 语言中最重要的部分之一的：类型系统。

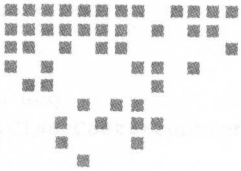
与 Java 相比，Kotlin 的类型系统更加简单一致，同时引入了可空类型、类型推断等特性，这些特性使得我们的代码更加安全、简洁。

另外，Kotlin 跟 Scala 类似，也提供了只读集合类型。关于只读集合类，我们将在下一章集合类中进行介绍。我们后面两章的主题分别是：集合类、泛型。

第二部分 *Part 2*

编程进阶

- 第 5 章　集合类
- 第 6 章　泛型
- 第 7 章　面向对象编程
- 第 8 章　函数式编程
- 第 9 章　轻量级线程：协程
- 第 10 章　Kotlin 与 Java 互操作



Chapter 5 第 5 章

集 合 类

本章介绍 Kotlin 标准库中的集合类：List、Set、Map。

Kotlin 中提供了不可变集合类与可变集合类。通过本章的学习，我们将了解到 Kotlin 是如何扩展的 Java 集合库，使得写代码更加简单容易。如果你熟悉 Scala 的集合库，你会发现 Kotlin 跟 Scala 集合类库的相似之处。

5.1 集合类的基本概念

曾经读过这样一段有趣的话，大致内容如下：

世间本无集合（只有数组）。

但有人想要，于是就用数组创造了集合类。

有人想要可以自动扩展容量的数组，于是有了 List。

有人想要元素不重复的数组，于是有了 Set。

有人想要有序的数组，于是有了 TreeSet、TreeList。

有人想要通过复杂对象（而不仅仅是简单的下表）来查找另一个对象的关联数组，于是有了 Map。

是的，几乎所有集合都是基于数组来实现的。所以，数组比集合要快。但是集合是对数组的高层次的抽象封装，集合的功能自然就比数组的功能要多得多。例如，数组是一种可读 / 可写数据结构，而使用集合提供的 ReadOnly 方法，我们就能够以只读方式来使用集合。

5.1.1 集合类是一种数据结构

在讲 Kotlin 的集合类之前，为了更加深刻理解为什么要有集合类，以及集合类到底是怎么一回事，让我们先来简单回顾一下编程的本质：

数据结构＋算法（信息的逻辑结构及其基本操作）

算法与数据的结构密切相关，算法无不依附于具体的数据结构，数据结构直接关系到算法的选择和效率。通常情况下，设计良好的数据结构可以大大简化算法的实现复杂度，同时可以提升存储效率。数据结构往往同高效的检索算法和索引技术相关（集合类中本质上就是这些东西）。

数据结构主要解决以下三个问题：

- 数据元素之间的逻辑关系。这些逻辑关系有：集合、线性结构、树形结构、图形结构等。
- 数据的物理结构。数据的逻辑结构在计算机存储空间的存放形式。数据的物理结构是数据结构在计算机中的映射。其具体实现的方法有：顺序（Sequence）、链接（Link）、索引（Index）、散列（Hash）等形式（这些就是常用的集合类了）。

  其中，顺序存储结构（数组、ArrayList）和链式存储结构（LinkedList）是我们常用的两种存储结构。顺序存储是使用元素在存储器中的相对位置来表示数据元素之间的逻辑关系；链式存储使用指示元素存储位置的指针（pointer）来表示数据元素之间的逻辑关系。
- 数据的处理运算。

我们可以把数据结构理解为是 ADT 的实现。数据结构就是现实问题模型的表达。

我们使用计算机编程来解决一个具体的现实问题时，通常需要经过下列几个步骤：

1）首先要从具体问题中抽象出一个适当的数学模型。

2）然后设计一个解此数学模型的算法。

3）最后编出程序、进行测试、修改直至得到最终解答。

这里面的寻求数学模型的过程，实质就是分析问题，从中提取操作的对象，并找出这些操作对象之间含有的关系的过程。最后，我们使用数学语言来表达这个模型。

这里的模型对应的就是数据结构。我们用计算机编程来解决问题的关键就是，设计出合适的数据结构（例如，用线性表、树、图等）和性能良好的算法。

编程语言中的集合类就像是一个百宝工具箱一样，来帮助我们解决问题。

5.1.2 集合类是 ADT

我们知道类的本质就是对象及其关系的抽象（abstraction），一个类通常有属性（数据结构）和行为（算法）。使用 OO 范式编程的大致过程有如下几个阶段：

划分对象→抽象类→将类组织成为层次化结构（继承和合成）→用类与实例进行设计和实现

数据抽象本质上讲就是我们解决现实问题的过程中，进行建立领域模型（Domain Model）的过程。

比如，在前一章节中，我们介绍的程序设计语言的类型系统，本质上就是一种数据抽象。由于计算机的结构和存储的限制（无法像人类大脑神经系统一样去认知识别，并解决现实问题），人类大脑在解决实际问题过程中，经常要计算整数、小数，要处理英文字符、中文字符，要持有对象（被操作的数据），要对这些对象进行诸如：查找、排序、修改、传递等操作。

我们把这些问题解决中最常用的数据结构以及其操作算法抽象成对应的类（例如：String、Array，集合类 List、Set、Map，等等），这样我们就可以极大地复用这些功能，而不需要我们自己来实现诸如：字符串、数组、列表、集合、映射等这些的数据结构。通常，这些最通用的数据结构都是现在编程语言中内置的了。

5.1.3 连续存储和离散存储

我们现在很少用抽象数据类型（Abstract Data Type，ADT）这个概念，其实这个概念是 OO 范式的前身，也是类的前身。使用 ADT 可以让我们更简单地描述现实世界。例如：用线性表描述学生成绩表，用树或图描述遗传关系等。

ADT 加上继承、重载和多态性就是现代 OOP 编程范式中的类的概念了。我们简称类为广义 ADT 的概念。

如果我们更加广义的来理解这里的 ADT 的思想，其实各种编程语言的 SDK API、所有的服务（IaaS、PaaS 和 SaaS 等）都是一种更加广义的 ADT。

5.2 Kotlin 集合类简介

集合类存放的都是对象的引用，而非对象本身，我们通常说的集合中的对象指的是集合中对象的引用（reference）。

Kotlin 的集合类分为：可变集合类（Mutable）与不可变集合类（Immutable）。

集合类型主要有 3 种：list（列表）、set（集）、和 map（映射）。

1. 列表

列表的主要特征是其对象以线性方式存储，没有特定顺序，只有一个开头和一个结尾，当然，它与根本没有顺序的集是不同的。列表在数据结构中可表现为：数组和向量、链表、堆栈、队列等。

2. 集

集（set）是最简单的一种集合，它的对象不按特定方式排序，只是简单地把对象加入集合中，就像往口袋里放东西。

对集中成员的访问和操作是通过集中对象的引用进行的，所以集中不能有重复对象。

集也有多种变体，可以实现排序等功能，如 TreeSet，它把对象添加到集中的操作将变

为按照某种比较规则将其插入到有序的对象序列中。它实现的是 SortedSet 接口，也就是加入了对象比较的方法。通过对集中的对象迭代，我们可以得到一个升序的对象集合。

3. 映射

映射（Map）与集或列表有明显区别，映射中每个项都是成对的。映射中存储的每个对象都有一个相关的关键字（Key）对象，关键字决定了对象在映射中的存储位置，检索对象时必须提供相应的关键字，就像在字典中查单词一样。关键字应该是唯一的。

关键字本身并不能决定对象的存储位置，它需要散列技术来处理，产生一个被称作散列码（hash code）的整数值，散列码通常用作一个偏置量，该偏置量是相对于分配给映射的内存区域起始位置的，由此确定“关键字 / 对象”对的存储位置。理想情况下，散列处理应该产生给定范围内均匀分布的值，而且每个关键字应得到不同的散列码。

5.3 List

List 接口继承于 Collection 接口，元素以线性方式存储，集合中可以存放重复对象。Kotlin 的 List 分为：不可变集合类 List（ReadOnly，Immutable）和可变集合类 MutableList（Read&Write，Mutable）。

List 类图结构如图 5-1 所示。

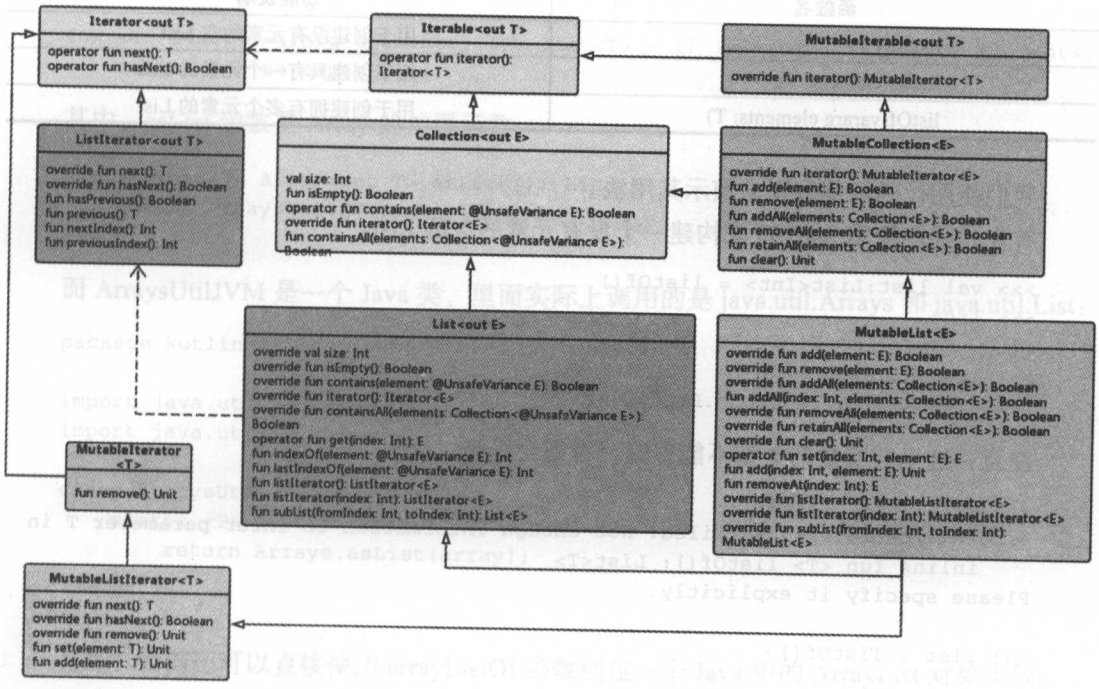


图 5-1 List 类架构

其中，Iterator 是所有容器类 Collection 的迭代器。迭代器（Iterator）模式又叫做游标（Cursor）模式。迭代器的定义为：提供一种方法访问一个容器对象中各个元素，而又不需暴露该对象的内部细节[⊖]。从定义可见，迭代器模式是为容器而生的。

5.3.1 创建不可变 List

我们可以使用 listOf 函数来构建一个不可变的 List（read-only，只读的 List）。它定义在 kotlin/collections/Collections.kt 里面。关于 listOf 这个构建函数有下面 3 个重载函数：

```
@kotlin.internal.InlineOnly
public inline fun <T> listOf(): List<T> = emptyList()

public fun <T> listOf(vararg elements: T): List<T> = if (elements.size > 0)
    elements.asList() else emptyList()

@JvmVersion
public fun <T> listOf(element: T): List<T> = java.util.Collections.
    singletonList(element)
```

这些函数创建的 List 都是只读的（也就是不可变的 immutable）、可序列化的，简单说明如表 5-1 所示

表 5-1 创建 List 的函数

函数名	功能说明
listOf()	用于创建没有元素的空 List
listOf(element: T)	用于创建只有一个元素的 List
listOf(vararg elements: T)	用于创建拥有多个元素的 List

我们使用代码示例分别来演示其用法。

首先，我们使用 listOf() 来构建一个没有元素的空的 List：

```
>>> val list:List<Int> = listOf()
>>> list
[]
>>> list::class
class kotlin.collections.EmptyList
```

注意，这里的变量的类型不能省略，否则会报错：

```
>>> val list = listOf()
error: type inference failed: Not enough information to infer parameter T in
    inline fun <T> listOf(): List<T>
Please specify it explicitly.

val list = listOf()
           ^
```

⊖ 参见《设计模式：可复用面向对象软件的基础》，由机械工业出版社出版。

因为这是一个泛型函数。关于泛型，我们将在下一章中介绍。

其中，EmptyList 是“internal object EmptyList”，这是 Kotlin 内部定义的一个默认空的 object List 类。

下面，我们再来创建一个只有 1 个元素的 List：

```
>>> val list = listOf(1)
>>> list::class
class java.util.Collections$SingletonList
```

我们可以看出，它实际上是调用 Java 的 java.util.Collections 里面的 singletonList 方法：

```
public static <T> List<T> singletonList(T o) {
    return new SingletonList<>(o);
}
```

我们再来创建一个有多个元素的 List：

```
>>> val list = listOf(0,1, 2, 3, 4, 5, 6,7,8,9)
>>> list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list::class
class java.util.Arrays$ArrayList
>>> list::class.java
```

它调用的是如下函数：

```
fun <T> listOf(vararg elements: T): List<T> = if (elements.size > 0) elements.
    asList() else emptyList()
```

其中，asList 函数是 Array 的扩展函数：

```
public fun <T> Array<out T>.asList(): List<T> {
    return ArraysUtilJVM.asList(this)
}
```

而 ArraysUtilJVM 是一个 Java 类，里面实际上调用的是 java.util.Arrays 和 java.util.List：

```
package kotlin.collections;

import java.util.Arrays;
import java.util.List;

class ArraysUtilJVM {
    static <T> List<T> asList(T[] array) {
        return Arrays.asList(array);
    }
}
```

另外，我们还可以直接使用 arrayListOf 函数创建一个 Java 中的 ArrayList 对象实例：

```
>>> val list = arrayListOf(0,1,2,3)
```

```
>>> list
[0, 1, 2, 3]
>>> list::class
class java.util.ArrayList
>>> val list = listOf(0,1, 2, 3, 4, 5, 6,7,8,9)
>>> list::class
class java.util.Arrays$ArrayList
```

这个函数定义在 libraries/stdlib/src/kotlin/collections/Collections.kt 类中：

```
@SinceKotlin("1.1")
@kotlin.internal.InlineOnly
public inline fun <T> arrayListOf(): ArrayList<T> = ArrayList()
```

同样的处理方式，这里的 ArrayList() 是 Java 中的 java.util.ArrayList 的类型别名：

```
@SinceKotlin("1.1") public typealias ArrayList<E> = java.util.ArrayList<E>
```

5.3.2 创建可变集合 MutableList

在 MutableList 中，除了继承 List 中的那些函数外，另外新增了 add/addAll、remove/removeAll/removeAt、set、clear、retainAll 等更新修改的操作函数：

```
override fun add(element: E): Boolean
override fun remove(element: E): Boolean
override fun addAll(elements: Collection<E>): Boolean
fun addAll(index: Int, elements: Collection<E>): Boolean
override fun removeAll(elements: Collection<E>): Boolean
override fun retainAll(elements: Collection<E>): Boolean
override fun clear(): Unit
operator fun set(index: Int, element: E): E
fun add(index: Int, element: E): Unit
fun removeAt(index: Int): E
override fun listIterator(): MutableListIterator<E>
override fun listIterator(index: Int): MutableListIterator<E>
override fun subList(fromIndex: Int, toIndex: Int): MutableList<E>
```

创建一个 MutableList 的对象实例跟 List 类似，前面加上前缀 mutable，代码示例如下：

```
>>> val list = mutableListOf(1, 2, 3)
>>> list
[1, 2, 3]
>>> list::class
class java.util.ArrayList
>>> val list2 = mutableListOf<Int>()
>>> list2
[]
>>> list2::class
class java.util.ArrayList
>>> val list3 = mutableListOf(1)
>>> list3
```

```
[1]
>>> list3::class
class java.util.ArrayList
```

我们可以看出，使用 mutableListOf 函数创建的可变集合类，实际上背后调用的是 java.util.ArrayList 类的相关方法。

另外，我们可以直接使用 Kotlin 封装的 arrayListOf 函数来创建一个 ArrayList：

```
>>> val list4 = arrayListOf(1, 2, 3)
>>> list4::class
class java.util.ArrayList
```

关于 Kotlin 中的 ArrayList 类型别名定义在 kotlin/collections/TypeAliases.kt 文件中：

```
@file:kotlin.jvm.JvmVersion

package kotlin.collections

@SinceKotlin("1.1") public typealias RandomAccess = java.util.RandomAccess


@SinceKotlin("1.1") public typealias ArrayList<E> = java.util.ArrayList<E>
@SinceKotlin("1.1") public typealias LinkedHashMap<K, V> = java.util.LinkedHashMap<K, V>
@SinceKotlin("1.1") public typealias HashMap<K, V> = java.util.HashMap<K, V>
@SinceKotlin("1.1") public typealias LinkedHashSet<E> = java.util.LinkedHashSet<E>
@SinceKotlin("1.1") public typealias HashSet<E> = java.util.HashSet<E>


//also @SinceKotlin("1.1")
internal typealias SortedSet<E> = java.util.SortedSet<E>
internal typealias TreeSet<E> = java.util.TreeSet<E>
```

如果我们已经有了一个不可变的 List，而现在想把它转换成可变的 List，可以直接调用转换函数 toMutableList：

```
val list = mutableListOf(1, 2, 3)
val mlist = list.toMutableList()
mlist.add(5)
```

5.3.3 遍历 List 元素

1. 使用 Iterator 迭代器

我们以集合 val list = listOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9) 为例，使用 Iterator 迭代器遍历列表所有元素的操作：

```
>>> val list = listOf(0,1, 2, 3, 4, 5, 6,7,8,9)
>>> list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> val iterator = list.iterator()
```

```
>>> iterator
java.util.AbstractList$Itr@438bad7c
>>> while(iterator.hasNext()){
... println(iterator.next())
... }
0
1
2
3
4
5
6
7
8
9
```

迭代器是一种设计模式，它是一个对象，可以遍历并选择序列中的对象，而开发人员不需要了解该序列的底层结构。迭代器通常称为“轻量级”对象，因为创建它的代价小。

Kotlin 中的 Iterator 功能比较简单，并且只能单向移动：

1）调用 iterator() 函数，容器返回一个 Iterator 实例。iterator() 函数是 kotlin.collections.Iterable 中的函数，被 Collection 继承。

2）调用 hasNext() 函数检查序列中是否还有元素。

3）第一次调用 Iterator 的 next() 函数时，它返回序列的第一个元素。依次向后递推，使用 next() 获得序列中的下一个元素。

当我们调用到最后一个元素，再次调用 next() 函数，会抛这个异常：java.util.NoSuchElementException。代码示例如下：

```
>>> val list = listOf(1,2,3)
>>> val iter = list.iterator()
>>> iter
java.util.AbstractList$Itr@3abfe845
>>> iter.hasNext()
true
>>> iter.next()
1
>>> iter.hasNext()
true
>>> iter.next()
2
>>> iter.hasNext()
true
>>> iter.next()
3
>>> iter.hasNext()
false
>>> iter.next()
java.util.NoSuchElementException
```

```
    at java.util.AbstractList$Itr.next(AbstractList.java:364)
```

我们可以看出，这里的 Iterator 的实现是在 AbstractList 中的内部类 IteratorImpl：

```
private open inner class IteratorImpl : Iterator<E> {
        protected var index = 0
        override fun hasNext(): Boolean = index < size
        override fun next(): E {
            if (!hasNext()) throw NoSuchElementException()
            return get(index++)
        }
    }
```

通过这个实现源码，我们可以更加清楚地明白 Iterator 的工作原理。

其中，NoSuchElementException() 这个类是 java.util.NoSuchElementException 的类型别名：

```
@kotlin.SinceKotlin public typealias NoSuchElementException = java.util.
    NoSuchElementException
```

2. 使用 forEach 遍历 List 元素

forEach 函数定义如下：

```
@kotlin.internal.HidesMembers
public inline fun <T> Iterable<T>.forEach(action: (T) -> Unit): Unit {
    for (element in this) action(element)
}
```

它是 package kotlin.collections 包下面的 Iterable 的扩展内联函数，它的入参是一个函数类型：

```
action: (T) -> Unit
```

关于函数式编程，我们将在后面章节中学习。

这里的 forEach 是一个语法糖。实际上 forEach 在遍历 List 对象的时候，仍然使用的是 iterator 迭代器来进行循环遍历的，如下所示：

```
>>> val list = listOf(1,2,3)
>>> list
[1, 2, 3]
>>> list.forEach{
... println(it)
... }
1
2
3
```

当参数只有一个函数的时候，括号可以省略不写。

也就是说，这里面的 forEach 函数调用的写法，实际上与下面的写法等价：

```
list.forEach({
    println(it)
})
```

我们甚至还可以直接这样写：

```
>>> list.forEach(::println)
```

其中，:: 是函数引用符。

5.3.4 List 元素操作函数

对 List 元素进行操作的函数有很多，请参见表 5-2。

表 5-2 List 元素操作函数

操作函数	功能说明
add、remove	添加、删除
retainAll	取两个集合交集
contains(element: T): Boolean	判断集合中是否有指定元素
elementAt(index: Int): T	查找下标对应的元素，如果下标越界抛出异常
first()	返回集合第 1 个元素，如果是空集抛出异常
indexOf(element: T): Int	返回指定下标的元素，没有则返回 -1
indexOfFirst(predicate: (T) -> Boolean): Int	返回第一个符合条件的元素下标，没有则返回 -1
indexOfLast(predicate: (T) -> Boolean): Int	返回最后一个符合条件的元素下标，没有则返回 -1
last()	返回集合最后一个元素，空集则抛出异常
last(predicate: (T) -> Boolean): T	返回符合条件的最后一个元素，否则就抛出异常 NoSuchElementException
lastIndexOf(element: T): Int	返回符合条件的最后一个元素，否则就返回 -1
single(): T	该集合如果只有 1 个元素，则返回该元素。否则，抛出异常
single(predicate: (T) -> Boolean): T	返回符合条件的单个元素，如果没有则抛出异常 NoSuchElementException，或超过一个的抛异常 IllegalArgumentException

1. add/remove 操作

这两个添加、删除操作函数是 MutableList 里面的，与 Java 中的集合类操作类似。

创建一个可变集合：

```
>>> val mutableList = mutableListOf(1,2,3)
>>> mutableList
[1, 2, 3]
```

向集合中添加一个元素：

```
>>> mutableList.add(4)
true
```

```
>>> mutableList
[1, 2, 3, 4]
```

在下标为 0 的位置添加元素 0：

```
>>> mutableList.add(0,0)
>>> mutableList
[0, 1, 2, 3, 4]
```

删除元素 1：

```
>>> mutableList.remove(1)
true
>>> mutableList
[0, 2, 3, 4]
>>> mutableList.remove(1)
false
```

删除下标为 1 的元素：

```
>>> mutableList.removeAt(1)
2
>>> mutableList
[0, 3, 4]
```

删除子集合：

```
>>> mutableList.removeAll(listOf(3,4))
true
>>> mutableList
[0]
```

添加子集合：

```
>>> mutableList.addAll(listOf(1,2,3))
true
>>> mutableList
[1, 2, 3]
```

2. set/clear 操作

更新设置下标 0 的元素值为 100：

```
>>> mutableList.set(0,100)
0
>>> mutableList
[100]
```

清空集合：

```
>>> mutableList.clear()
>>> mutableList
[]
```

把可变集合转为不可变集合：

```
>>> mutableList.toList()
[1, 2, 3]
```

3. retainAll

取两个集合交集：

```
>>> val mlist1 = mutableListOf(1,2,3,4,5,6)
>>> val mlist2 = mutableListOf(3,4,5,6,7,8,9)
>>> mlist1.retainAll(mlist2)
true
>>> mlist1
[3, 4, 5, 6]
```

4. contains(element: T): Boolean

判断集合中是否有指定元素，有就返回 true，否则返回 false。

代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7)
>>> list.contains(1)
true
```

5. elementAt(index: Int): T

查找下标对应的元素，如果下标越界，会抛出异常 IndexOutOfBoundsException。

代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7)
>>> list.elementAt(6)
7
>>> list.elementAt(7)
java.lang.ArrayIndexOutOfBoundsException: 7
    at java.util.Arrays$ArrayList.get(Arrays.java:3841)
```

另外，针对越界的处理，还有下面两个函数：

elementAtOrElse(index: Int, defaultValue: (Int) -> T): T：查找下标对应元素，如果越界会根据方法返回默认值：

```
>>> list.elementAtOrElse(7,{0})
0
>>> list.elementAtOrElse(7,{10})
10
```

elementAtOrNull(index: Int): T?：查找下标对应元素，如果越界就返回 null：

```
>>> list.elementAtOrNull(7)
null
```

6. first()

返回集合第 1 个元素，如果是空集，抛出异常 NoSuchElementException。

```
>>> val list = listOf(1,2,3)
>>> list.first()
1
>>> val emptyList = listOf<Int>()
>>> emptyList.first()
java.util.NoSuchElementException: List is empty.
    at kotlin.collections.CollectionsKt___CollectionsKt.first(_Collections.kt:178)
```

对应的有针对异常处理的函数 firstOrNull(): T?：

```
>>> emptyList.firstOrNull()
null
first(predicate: (T) -> Boolean): T
```

返回符合条件的第一个元素，没有则抛异常 NoSuchElementException。

```
>>> val list = listOf(1,2,3)
>>> list.first({it%2==0})
2
>>> list.first({it>100})
java.util.NoSuchElementException: Collection contains no element matching the predicate.
```

对应的有针对异常处理的函数 firstOrNull(predicate: (T) -> Boolean): T?，返回符合条件的第一个元素，没有就返回 null：

```
>>> list.firstOrNull({it>100})
null
```

7. indexOf(element: T): Int

返回指定下标的元素，没有就返回 -1：

```
>>> val list = listOf("a","b","c")
>>> list.indexOf("c")
2
>>> list.indexOf("x")
-1
```

8. indexOfFirst(predicate: (T) -> Boolean): Int

返回第一个符合条件的元素下标，没有就返回 -1：

```
>>> val list = listOf("abc","xyz","xjk","pqk")
>>> list.indexOfFirst({it.contains("x")})
1
>>> list.indexOfFirst({it.contains("k")})
2
>>> list.indexOfFirst({it.contains("e")})
-1
```

9. indexOfLast(predicate: (T) -> Boolean): Int

返回最后一个符合条件的元素下标，没有就返回 -1：

```
>>> val list = listOf("abc","xyz","xjk","pqk")
>>> list.indexOfLast({it.contains("x")})
2
>>> list.indexOfLast({it.contains("k")})
3
>>> list.indexOfLast({it.contains("e")})
-1
```

10. last()

返回集合最后一个元素，空集则抛出异常 NoSuchElementException：

```
>>> val list = listOf(1,2,3,4,7,5,6,7,8)
>>> list.last()
8
>>> val emptyList = listOf<Int>()
>>> emptyList.last()
java.util.NoSuchElementException: List is empty.
    at kotlin.collections.CollectionsKt___CollectionsKt.last(_Collections.kt:340)
```

11. last(predicate: (T) -> Boolean): T

返回符合条件的最后一个元素，没有就抛出异常 NoSuchElementException：

```
>>> val list = listOf(1,2,3,4,7,5,6,7,8)
>>> list.last({it==7})
7
>>> list.last({it>10})
java.util.NoSuchElementException: List contains no element matching the predicate.
```

对应的针对越界处理的 lastOrNull 函数：返回符合条件的最后一个元素，没有则返回 null：

```
>>> list.lastOrNull({it>10})
null
```

12. lastIndexOf(element: T): Int

返回符合条件的最后一个元素，没有就返回 -1：

```
>>> val list = listOf("abc","dfg","jkl","abc","bbc","wer")
>>> list.lastIndexOf("abc")
3
```

13. single(): T

该集合如果只有 1 个元素，则返回该元素。否则，抛出异常：

```
>>> val list = listOf(1)
>>> list.single()
```

```
1

>>> val list = listOf(1,2)
>>> list.single()
java.lang.IllegalArgumentException: List has more than one element.
    at kotlin.collections.CollectionsKt___CollectionsKt.single(_Collections.kt:471)

>>> val list = listOf<Int>()

>>> list.single()
java.util.NoSuchElementException: List is empty.
    at kotlin.collections.CollectionsKt___CollectionsKt.single(_Collections.kt:469)
```

14. single(predicate: (T) -> Boolean): T

返回符合条件的单个元素，如果没有符合的则抛出异常 NoSuchElementException，或超过一个的抛异常 IllegalArgumentException：

```
>>> val list = listOf(1,2,3,4,7,5,6,7,8)
>>> list.single({it==1})
1
>>> list.single({it==7})
java.lang.IllegalArgumentException: Collection contains more than one matching element.

>>> list.single({it==10})
java.util.NoSuchElementException: Collection contains no element matching the predicate.
```

对应的针对异常处理的函数 singleOrNull: 返回符合条件的单个元素，如有没有符合或超过一个，返回 null：

```
>>> list.singleOrNull({it==7})
null
>>> list.singleOrNull({it==10})
null
```

5.3.5 List 集合类的基本运算函数

List 集合类的基本运算函数算子如表 5-3 所示。

表 5-3 List 集合类的基本运算函数及功能说明

函数算子	功能说明
any()	判断集合至少有一个元素
any(predicate: (T) -> Boolean)	判断集合中是否有满足条件的元素
all(predicate: (T) -> Boolean)	判断集合中的元素是否都满足条件
none()	判断集合无元素
none(predicate: (T) -> Boolean)	判断集合中所有元素都不满足条件
count()	计算集合中元素的个数
count(predicate: (T) -> Boolean)	计算集合中满足条件的元素的个数

（续）

函数算子	功能说明
reduce	从第一项到最后一项进行累计运算
reduceRight	从最后一项到第一项进行累计运算
fold(initial: R, operation: (acc: R, T) -> R): R	带初始值的 reduce
forEach(action: (T) -> Unit): Unit	循环遍历元素，元素是 it
forEachIndexed	带 index（下标）的元素遍历
max、min	查询最大、最小的元素，空集则返回 null
maxBy(selector: (T) -> R): T?、minBy(selector: (T) -> R): T?	获取函数映射结果的最大值、最小值对应的那个元素的值，如果没有则返回 null
sumBy(selector: (T) -> Int): Int	获取函数映射值的总和

1. any()

判断集合至少有一个元素，这个函数定义如下：

```
public fun <T> Iterable<T>.any(): Boolean {
    for (element in this) return true
    return false
}
```

如果该集合至少有一个元素，返回 true；否则返回 false。代码示例：

```
>>> val emptyList = listOf<Int>()
>>> emptyList.any()
false
>>> val list1 = listOf(1)
>>> list1.any()
true
```

2. any(predicate: (T) -> Boolean)

判断集合中是否有满足条件的元素，这个函数定义如下：

```
public inline fun <T> Iterable<T>.any(predicate: (T) -> Boolean): Boolean {
    for (element in this) if (predicate(element)) return true
    return false
}
```

如果该集合中至少有一个元素匹配谓词函数参数 predicate: (T) -> Boolean，返回 true；否则返回 false。代码示例：

```
>>> val list = listOf(1, 2, 3)
>>> list.any()              // 至少有 1 个元素
true
>>> list.any({it%2==0})     // 元素 2 满足 {it%2==0}
true
>>> list.any({it>4})        // 没有元素满足 {it>4}
false
```

3. all(predicate: (T) -> Boolean)

判断集合中的元素是否都满足条件，函数定义：

```
public inline fun <T> Iterable<T>.all(predicate: (T) -> Boolean): Boolean {
    for (element in this) if (!predicate(element)) return false
    return true
}
```

当且仅当该集合中所有元素都满足条件时，返回 true；否则都返回 false。代码示例：

```
>>> val list = listOf(0,2,4,6,8)
>>> list.all({it%2==0})
true
>>> list.all({it>2})
false
```

4. none()

判断集合无元素，函数定义：

```
public fun <T> Iterable<T>.none(): Boolean {
    for (element in this) return false
    return true
}
```

如果该集合没有任何元素，返回 true；否则返回 false。代码示例：

```
>>> val list = listOf<Int>()
>>> list.none()
true
```

5. none(predicate: (T) -> Boolean)

判断集合中所有元素都不满足条件，函数定义：

```
public inline fun <T> Iterable<T>.none(predicate: (T) -> Boolean): Boolean {
    for (element in this) if (predicate(element)) return false
    return true
}
```

当且仅当集合中所有元素都不满足条件时返回 true；否则返回 false。代码示例：

```
>>> val list = listOf(0,2,4,6,8)
>>> list.none({it%2==1})
true
>>> list.none({it>0})
false
```

6. count()

计算集合中元素的个数，函数定义：

```
public fun <T> Iterable<T>.count(): Int {
```

```
    var count = 0
    for (element in this) count++
    return count
}
```

代码示例：

```
>>> val list = listOf(0,2,4,6,8,9)
>>> list.count()
6
```

7. count(predicate: (T) -> Boolean)

计算集合中满足条件的元素的个数，函数定义：

```
public inline fun <T> Iterable<T>.count(predicate: (T) -> Boolean): Int {
    var count = 0
    for (element in this) if (predicate(element)) count++
    return count
}
```

代码示例：

```
>>> val list = listOf(0,2,4,6,8,9)
>>> list.count()
6
>>> list.count({it%2==0})
5
```

8. reduce

从第一项到最后一项进行累计运算，函数定义：

```
public inline fun <S, T: S> Iterable<T>.reduce(operation: (acc: S, T) -> S): S {
    val iterator = this.iterator()
    if (!iterator.hasNext()) throw UnsupportedOperationException("Empty collection
        can't be reduced.")
    var accumulator: S = iterator.next()
    while (iterator.hasNext()) {
        accumulator = operation(accumulator, iterator.next())
    }
    return accumulator
}
```

首先把第一个元素赋值给累加子 accumulator，然后逐次向后取元素累加，新值继续赋值给累加子 accumulator = operation(accumulator, iterator.next())，以此类推。最后返回累加子的值。代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7,8,9)
>>> list.reduce({sum, next->sum+next})
45
>>> list.reduce({sum, next->sum*next})
```

```
362880
>>> val list = listOf("a","b","c")
>>> list.reduce({total, s->total+s})
abc
```

9. reduceRight

从最后一项到第一项进行累计运算，函数定义：

```
public inline fun <S, T: S> List<T>.reduceRight(operation: (T, acc: S) -> S): S {
    val iterator = listIterator(size)
    if (!iterator.hasPrevious())
        throw UnsupportedOperationException("Empty list can't be reduced.")
    var accumulator: S = iterator.previous()
    while (iterator.hasPrevious()) {
        accumulator = operation(iterator.previous(), accumulator)
    }
    return accumulator
}
```

从函数的定义 accumulator = operation(iterator.previous(), accumulator)，可以看出，从右边累计运算的累加子是放在后面的。代码示例：

```
>>> val list = listOf("a","b","c")
>>> list.reduceRight({total, s -> s+total})
cba
```

如果我们位置放错了，会输出下面的结果：

```
>>> list.reduceRight({total, s -> total+s})
abc
```

10. fold(initial: R, operation: (acc: R, T) -> R): R

带初始值的 reduce，函数定义：

```
public inline fun <T, R> Iterable<T>.fold(initial: R, operation: (acc: R, T) -> R): R {
    var accumulator = initial
    for (element in this) accumulator = operation(accumulator, element)
    return accumulator
}
```

从函数的定义可以看出，fold 函数给累加子赋了初始值 initial。代码示例：

```
>>> val list=listOf(1,2,3,4)
>>> list.fold(100,{total, next -> next + total})
110
```

foldRight 和 reduceRight 类似，有初始值。函数定义：

```
public inline fun <T, R> List<T>.foldRight(initial: R, operation: (T, acc: R) -> R): R {
    var accumulator = initial
    if (!isEmpty()) {
```

```
        val iterator = listIterator(size)
        while (iterator.hasPrevious()) {
            accumulator = operation(iterator.previous(), accumulator)
        }
    }
    return accumulator
}
```

代码示例：

```
>>> val list = listOf("a","b","c")
>>> list.foldRight("xyz",{s, pre -> pre + s})
xyzcba
```

11. forEach(action: (T) -> Unit): Unit

循环遍历元素，元素是 it，我们在前文已经讲述，参见 5.3.4 节。再写个代码示例：

```
>>> val list = listOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> list.forEach { value -> if (value > 7) println(value) }
8
9
```

12. forEachIndexed

带 index（下标）的元素遍历，函数定义：

```
public inline fun <T> Iterable<T>.forEachIndexed(action: (index: Int, T) -> Unit): Unit {
    var index = 0
    for (item in this) action(index++, item)
}
```

代码示例：

```
>>> val list = listOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> list.forEachIndexed { index, value -> if (value > 8) println("value of index
    $index is $value, greater than 8") }

value of index 9 is 9, greater than 8
```

13. max、min

查询最大、最小的元素，空集则返回 null，max 函数定义：

```
public fun <T : Comparable<T>> Iterable<T>.max(): T? {
    val iterator = iterator()
    if (!iterator.hasNext()) return null
    var max = iterator.next()
    while (iterator.hasNext()) {
        val e = iterator.next()
        if (max < e) max = e
    }
    return max
}
```

返回集合中最大的元素。代码示例：

```
>>> val list = listOf(1,2,3)
>>> list.max()
3
>>> val list = listOf("a","b","c")
>>> list.max()
c
```

min 函数定义：

```
public fun <T : Comparable<T>> Iterable<T>.min(): T? {
    val iterator = iterator()
    if (!iterator.hasNext()) return null
    var min = iterator.next()
    while (iterator.hasNext()) {
        val e = iterator.next()
        if (min > e) min = e
    }
    return min
}
```

返回集合中的最小元素。代码示例：

```
>>> val list = listOf(1,2,3)
>>> list.min()
1
>>> val list = listOf("a","b","c")
>>> list.min()
a
```

在 Kotlin 中，比较字符串的大小很有意思，我们直接通过代码示例来学习一下：

```
>>> "c" > "a"
true
>>> "abd" > "abc"
true
>>> "abd" > "abcd"
true
>>> "abd" > "abcdefg"
true
```

可以看出，比较字符串的大小是按照对应的下标字符进行的。

另外，布尔值的比较是 true 大于 false：

```
>>> true > false
true
```

14. maxBy(selector: (T) -> R): T? 、minBy(selector: (T) -> R): T?

获取函数映射结果的最大值、最小值对应的那个元素的值，如果没有则返回 null。函数定义：

```
public inline fun <T, R : Comparable<R>> Iterable<T>.maxBy(selector: (T) -> R): T? {
    val iterator = iterator()
    if (!iterator.hasNext()) return null
    var maxElem = iterator.next()
    var maxValue = selector(maxElem)
    while (iterator.hasNext()) {
        val e = iterator.next()
        val v = selector(e)
        if (maxValue < v) {
            maxElem = e
            maxValue = v
        }
    }
    return maxElem
}
```

也就是说，不直接比较集合元素的大小，而是以集合元素为入参的函数 selector: (T) -> R 返回值来比较大小，最后返回此元素的值（注意，不是对应的 selector 函数的返回值）。有点像数学里的求函数最值问题：

给定函数 y = f(x)，求 max f(x) 的 x 的值。代码示例：

```
>>> val list = listOf(100,-500,300,200)
>>> list.maxBy({it})
300
>>> list.maxBy({it*(1-it)})
100
>>> list.maxBy({it*it})
-500
```

对应的 minBy 是获取函数映射后返回结果的最小值所对应那个元素的值，如果没有则返回 null。代码示例：

```
>>> val list = listOf(100,-500,300,200)
>>> list.minBy({it})
-500
>>> list.minBy({it*(1-it)})
-500
>>> list.minBy({it*it})
100
```

15. sumBy(selector: (T) -> Int): Int

获取函数映射值的总和，函数定义：

```
public inline fun <T> Iterable<T>.sumBy(selector: (T) -> Int): Int {
    var sum: Int = 0
    for (element in this) {
        sum += selector(element)
    }
    return sum
}
```

可以看出，这个 sumBy 函数算子，累加器 sum 初始值为 0，返回值是 Int。它的入参 selector 是一个函数类型 (T) -> Int，也就是说，这个 selector 也是返回 Int 类型的函数。代码示例：

```
>>> val list = listOf(1,2,3,4)
>>> list.sumBy({it})
10
>>> list.sumBy({it*it})
30
```

类型错误反例：

```
>>> val list = listOf("a","b","c")
>>> list.sumBy({it})
error: type inference failed: inline fun <T> Iterable<T>.sumBy(selector: (T) ->
    Int): Int
cannot be applied to
receiver: List<String>  arguments: ((String) -> String)

list.sumBy({it})
     ^
error: type mismatch: inferred type is (String) -> String but (String) -> Int
    was expected
list.sumBy({it})
           ^
```

5.3.6 过滤操作函数

常用的过滤操作函数算子如表 5-4 所示。

表 5-4 常用过滤操作函数及功能说明

函数算子	功能说明
take(n: Int): List<T>	挑出该集合前 n 个元素的子集合
takeWhile(predicate: (T) -> Boolean): List<T>	挑出满足条件的元素的子集合
takeLast	挑出后 n 个元素的子集合
takeLastWhile(predicate: (T) -> Boolean)	从最后开始挑出满足条件元素的子集合
drop(n: Int)	去除前 n 个元素返回剩下的元素的子集合
dropWhile(predicate: (T) -> Boolean)	去除满足条件的元素返回剩下的元素的子集合
dropLast(n: Int)	从最后去除 n 个元素
dropLastWhile(predicate: (T) -> Boolean)	从最后去除满足条件的元素
slice(indices: IntRange)	取开始下标至结束下标元素子集合
slice(indices: Iterable<Int>)	返回指定下标的元素子集合
filterTo(destination: C, predicate: (T) -> Boolean)	过滤出满足条件的元素并赋值给 destination
filter(predicate: (T) -> Boolean)	过滤出满足条件的元素组成的子集合
filterNot(predicate: (T) -> Boolean)	过滤出所有不满足条件的元素
filterNotNull()	过滤出非 null 元素

1. take(n: Int): List<T>

挑出该集合前 n 个元素的子集合。函数定义：

```
public fun <T> Iterable<T>.take(n: Int): List<T> {
    require(n >= 0) { "Requested element count $n is less than zero." }
    if (n == 0) return emptyList()
    if (this is Collection<T>) {
        if (n >= size) return toList()
        if (n == 1) return listOf(first())
    }
    var count = 0
    val list = ArrayList<T>(n)
    for (item in this) {
        if (count++ == n)
            break
        list.add(item)
    }
    return list.optimizeReadOnlyList()
}
```

如果 n 等于 0，返回空集；如果 n 大于集合 size，返回该集合。代码示例：

```
>>> val list = listOf("a","b","c")
>>> list
[a, b, c]
>>> list.take(2)
[a, b]
>>> list.take(10)
[a, b, c]
>>> list.take(0)
[]
```

2. takeWhile(predicate: (T) -> Boolean): List<T>

挑出满足条件的元素的子集合。函数定义：

```
public inline fun <T> Iterable<T>.takeWhile(predicate: (T) -> Boolean): List<T> {
    val list = ArrayList<T>()
    for (item in this) {
        if (!predicate(item))
            break
        list.add(item)
    }
    return list
}
```

从第一个元素开始，判断是否满足 predicate 为 true，如果满足条件的元素就丢到返回 ArrayList 中。只要遇到任何一个元素不满足条件，就结束循环，返回 list。代码示例：

```
>>> val list = listOf(1,2,4,6,8,9)
>>> list.takeWhile({it%2==0})
```

```
[]
>>> list.takeWhile({it%2==1})
[1]

>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.takeWhile({it%2==0})
[2, 4, 6, 8]
```

3. takeLast

挑出后 n 个元素的子集合。函数定义：

```
public fun <T> List<T>.takeLast(n: Int): List<T> {
    require(n >= 0) { "Requested element count $n is less than zero." }
    if (n == 0) return emptyList()
    val size = size
    if (n >= size) return toList()
    if (n == 1) return listOf(last())
    val list = ArrayList<T>(n)
    if (this is RandomAccess) {
        for (index in size - n .. size - 1)
            list.add(this[index])
    } else {
        for (item in listIterator(n))
            list.add(item)
    }
    return list
}
```

从集合倒数 n 个元素起，取出到最后一个元素的子集合。如果传入 0，返回空集。如果传入 n 大于集合 size，返回整个集合。如果传入负数，直接抛出 IllegalArgumentException。代码示例：

```
>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.takeLast(0)
[]
>>> list.takeLast(3)
[11, 12, 16]
>>> list.takeLast(100)
[2, 4, 6, 8, 9, 11, 12, 16]
>>> list.takeLast(-1)
java.lang.IllegalArgumentException: Requested element count -1 is less than zero.
    at kotlin.collections.CollectionsKt___CollectionsKt.takeLast(_Collections.kt:734)
```

4. takeLastWhile(predicate: (T) -> Boolean)

从最后开始挑出满足条件元素的子集合。函数定义：

```
public inline fun <T> List<T>.takeLastWhile(predicate: (T) -> Boolean): List<T> {
    if (isEmpty())
        return emptyList()
    val iterator = listIterator(size)
```

```
    while (iterator.hasPrevious()) {
        if (!predicate(iterator.previous())) {
            iterator.next()
            val expectedSize = size - iterator.nextIndex()
            if (expectedSize == 0) return emptyList()
            return ArrayList<T>(expectedSize).apply {
                while (iterator.hasNext())
                    add(iterator.next())
            }
        }
    }
    return toList()
}
```

反方向取满足条件的元素，遇到不满足的元素，直接终止循环，并返回子集合。代码示例：

```
>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.takeLastWhile({it%2==0})
[12, 16]
```

5. drop(n: Int)

去除前 n 个元素返回剩下的元素的子集合。函数定义：

```
public fun <T> Iterable<T>.drop(n: Int): List<T> {
    require(n >= 0) { "Requested element count $n is less than zero." }
    if (n == 0) return toList()
    val list: ArrayList<T>
    if (this is Collection<*>) {
        val resultSize = size - n
        if (resultSize <= 0)
            return emptyList()
        if (resultSize == 1)
            return listOf(last())
        list = ArrayList<T>(resultSize)
        if (this is List<T>) {
            if (this is RandomAccess) {
                for (index in n..size - 1)
                    list.add(this[index])
            } else {
                for (item in listIterator(n))
                    list.add(item)
            }
            return list
        }
    }
    else {
        list = ArrayList<T>()
    }
```

```
    var count = 0
    for (item in this) {
        if (count++ >= n) list.add(item)
    }
    return list.optimizeReadOnlyList()
}
```

代码示例：

```
>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.drop(5)
[11, 12, 16]
>>> list.drop(100)
[]
>>> list.drop(0)
[2, 4, 6, 8, 9, 11, 12, 16]
>>> list.drop(-1)
java.lang.IllegalArgumentException: Requested element count -1 is less than zero.
    at kotlin.collections.CollectionsKt___CollectionsKt.drop(_Collections.kt:538)
```

6. dropWhile(predicate: (T) -> Boolean)

去除满足条件的元素返回剩下的元素的子集合。函数定义：

```
public inline fun <T> Iterable<T>.dropWhile(predicate: (T) -> Boolean): List<T> {
    var yielding = false
    val list = ArrayList<T>()
    for (item in this)
        if (yielding)
            list.add(item)
        else if (!predicate(item)) {
            list.add(item)
            yielding = true
        }
    return list
}
```

去除满足条件的元素，当遇到一个不满足条件的元素时，中止操作，返回剩下的元素子集合。代码示例：

```
>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.dropWhile({it%2==0})
[9, 11, 12, 16]
```

7. dropLast(n: Int)

从最后去除 n 个元素，函数定义：

```
public fun <T> List<T>.dropLast(n: Int): List<T> {
    require(n >= 0) { "Requested element count $n is less than zero." }
    return take((size - n).coerceAtLeast(0))
}
```

代码示例：

```
>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.dropLast(3)
[2, 4, 6, 8, 9]
>>> list.dropLast(100)
[]
>>> list.dropLast(0)
[2, 4, 6, 8, 9, 11, 12, 16]
>>> list.dropLast(-1)
java.lang.IllegalArgumentException: Requested element count -1 is less than zero.
    at kotlin.collections.CollectionsKt___CollectionsKt.dropLast(_Collections.kt:573)
```

8. dropLastWhile(predicate: (T) -> Boolean)

从最后去除满足条件的元素，函数定义：

```
public inline fun <T> List<T>.dropLastWhile(predicate: (T) -> Boolean): List<T> {
    if (!isEmpty()) {
        val iterator = listIterator(size)
        while (iterator.hasPrevious()) {
            if (!predicate(iterator.previous())) {
                return take(iterator.nextIndex() + 1)
            }
        }
    }
    return emptyList()
}
```

代码示例：

```
>>> val list = listOf(2,4,6,8,9,11,12,16)
>>> list.dropLastWhile({it%2==0})
[2, 4, 6, 8, 9, 11]
```

9. slice(indices: IntRange)

取开始下标至结束下标元素子集合。函数定义：

```
public fun <T> List<T>.slice(indices: IntRange): List<T> {
    if (indices.isEmpty()) return listOf()
    return this.subList(indices.start, indices.endInclusive + 1).toList()
}
```

代码示例：

```
val list = listOf(2,4,6,8,9,11,12,16)
>>> list
[2, 4, 6, 8, 9, 11, 12, 16]
>>> list.slice(1..3)
[4, 6, 8]
>>> list.slice(2..7)
```

```
[6, 8, 9, 11, 12, 16]
>>> list
[2, 4, 6, 8, 9, 11, 12, 16]
>>> list.slice(1..3)
[4, 6, 8]
>>> list.slice(2..7)
[6, 8, 9, 11, 12, 16]
```

10. slice(indices: Iterable<Int>)

返回指定下标的元素子集合。函数定义：

```
public fun <T> List<T>.slice(indices: Iterable<Int>): List<T> {
    val size = indices.collectionSizeOrDefault(10)
    if (size == 0) return emptyList()
    val list = ArrayList<T>(size)
    for (index in indices) {
        list.add(get(index))
    }
    return list
}
```

这个函数从签名上看不是那么简单直接。从函数的定义看，这里的 indices 是当做原来集合的下标来使用的。代码示例：

```
>>> list
[2, 4, 6, 8, 9, 11, 12, 16]
>>> list.slice(listOf(2,4,6))
[6, 9, 12]
```

我们可以看出，这里是取出下标为 2、4、6 的元素，而不是直观理解上的，去掉元素 2、4、6。

11. filterTo(destination: C, predicate: (T) -> Boolean)

过滤出满足条件的元素并赋值给 destination。函数定义：

```
public inline fun <T, C : MutableCollection<in T>> Iterable<T>.filterTo(destination:
    C, predicate: (T) -> Boolean): C {
    for (element in this) if (predicate(element)) destination.add(element)
    return destination
}
```

把满足过滤条件的元素组成的子集合赋值给入参 destination。代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7)
>>> val dest = mutableListOf<Int>()
>>> list.filterTo(dest,{it>3})
[4, 5, 6, 7]
>>> dest
[4, 5, 6, 7]
```

12. filter(predicate: (T) -> Boolean)

过滤出满足条件的元素组成的子集合。函数定义：

```
public inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T> {
    return filterTo(ArrayList<T>(), predicate)
}
```

相对于filterTo函数，filter函数更加简单易用。从源码我们可以看出，filter函数直接调用的filterTo(ArrayList<T>(), predicate)，其中入参destination被直接默认赋值为ArrayList<T>()。代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7)
>>> list.filter({it>3})
[4, 5, 6, 7]
```

另外，还有两个常用的过滤函数：

- filterNot(predicate: (T) -> Boolean)，用来过滤所有不满足条件的元素。
- filterNotNull() 过滤掉null元素。

5.3.7 映射操作函数

常用的映射操作函数如表5-5所示。

表5-5 映射操作符函数及功能说明

函 数	功能说明
map(transform: (T) -> R): List<R>	将集合中的元素通过转换函数transform映射后的结果，存到一个集合中返回
mapIndexed(transform: (kotlin.Int, T) -> R)	转换函数transform中带有下标参数。也就是说我们可以同时使用下标和元素的值来进行转换。其中，第一个参数是Int类型的下标
mapNotNull(transform: (T) -> R?)	遍历集合每个元素，得到通过函数算子transform映射之后的值，剔除掉这些值中的null，返回一个无null元素的集合
flatMap(transform: (T) -> Iterable<R>): List<R>	在原始集合的每个元素上调用transform转换函数，得到的映射结果组成的单个列表

1. map(transform: (T) -> R): List<R>

将集合中的元素通过转换函数transform映射后的结果，存到一个集合中返回，代码如下：

```
>>> val list = listOf(1,2,3,4,5,6,7)
>>> list.map({it})
[1, 2, 3, 4, 5, 6, 7]
>>> list.map({it*it})
[1, 4, 9, 16, 25, 36, 49]
>>> list.map({it+10})
[11, 12, 13, 14, 15, 16, 17]
```

这个函数内部调用的是：

```
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    return mapTo(ArrayList<R>(collectionSizeOrDefault(10)), transform)
}
```

这里的 mapTo 函数定义如下：

```
public inline fun <T, R, C : MutableCollection<in R>> Iterable<T>.mapTo (destination:
    C, transform: (T) -> R): C {
    for (item in this)
        destination.add(transform(item))
    return destination
}
```

我们可以看出，这个 map 实现的原理是循环遍历原集合中的元素，并把通过 transform 映射后的结果放到一个新的 destination 集合中，并返回 destination。

2. mapIndexed(transform: (kotlin. Int, T) -> R)

转换函数 transform 中带有下标参数。也就是说我们可以同时使用下标和元素的值来进行转换。其中，第一个参数是 Int 类型的下标。代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7)
>>> list.mapIndexed({index,it -> index*it})
[0, 2, 6, 12, 20, 30, 42]
```

3. mapNotNull(transform: (T) -> R?)

遍历集合每个元素，得到通过函数算子 transform 映射之后的值，剔除掉这些值中的 null，返回一个无 null 元素的集合。代码示例：

```
>>> val list = listOf("a","b",null,"x",null,"z")
>>> list.mapNotNull({it})
[a, b, x, z]
```

这个函数内部实现是调用的 mapNotNullTo 函数：

```
public inline fun <T, R : Any, C : MutableCollection<in R>> Iterable<T>.
    mapNotNullTo(destination: C, transform: (T) -> R?): C {
    forEach { element -> transform(element)?.let { destination.add(it) } }
    return destination
}
```

4. flatMap(transform: (T) -> Iterable<R>): List<R>

在原始集合的每个元素上调用 transform 转换函数，得到的映射结果组成的单个列表。为了更简单的理解这个函数，我们跟 map(transform: (T) -> R): List<R> 对比下。

首先看函数的各自的实现：

```
map:
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
```

```
    return mapTo(ArrayList<R>(collectionSizeOrDefault(10)), transform)
}

public inline fun <T, R, C : MutableCollection<in R>> Iterable<T>.mapTo(destination:
    C, transform: (T) -> R): C {
    for (item in this)
        destination.add(transform(item))
    return destination
}
flatMap:
public inline fun <T, R> Iterable<T>.flatMap(transform: (T) -> Iterable<R>): List<R> {
    return flatMapTo(ArrayList<R>(), transform)
}

public inline fun <T, R, C : MutableCollection<in R>> Iterable<T>.flatMapTo
    (destination: C, transform: (T) -> Iterable<R>): C {
    for (element in this) {
        val list = transform(element)
        destination.addAll(list)
    }
    return destination
}
```

可以看出，这两个函数主要区别在 transform 函数返回上。代码示例：

```
>>> val list = listOf("a","b","c")
>>> list.map({it->listOf(it+1,it+2,it+3)})
[[a1, a2, a3], [b1, b2, b3], [c1, c2, c3]]
>>> list.flatMap({it->listOf(it+1,it+2,it+3)})
[a1, a2, a3, b1, b2, b3, c1, c2, c3]
```

从代码运行结果我们可以看出，使用 map 是把 list 中的每一个元素都映射成一个 List-n，然后以这些 List-n 为元素，组成一个大的嵌套的 List 返回。而使用 flatMap 则是把 list 中的第一个元素映射成一个 List1，然后把第二个元素映射成的 List2 跟 List1 合并：List1.addAll(List2)，以此类推。最终返回一个“扁平的”(flat) List。

其实，这个 flatMap 的过程是 map + flatten 两个操作的组合。这个 flatten 函数定义如下：

```
public fun <T> Iterable<Iterable<T>>.flatten(): List<T> {
    val result = ArrayList<T>()
    for (element in this) {
        result.addAll(element)
    }
    return result
}
```

代码示例：

```
>>> val list = listOf("a","b","c")
>>> list.map({it->listOf(it+1,it+2,it+3)})
```

```
[[a1, a2, a3], [b1, b2, b3], [c1, c2, c3]]
>>> list.map({it->listOf(it+1,it+2,it+3)}).flatten()
[a1, a2, a3, b1, b2, b3, c1, c2, c3]
```

5.3.8 分组操作函数

常用的分组操作函数有两个，下面分别介绍。

1. groupBy(keySelector: (T) -> K): Map<K, List<T>>

将集合中的元素按照条件选择器 keySelector（是一个函数）分组，并返回 Map。代码示例：

```
>>> val words = listOf("a", "abc", "ab", "def", "abcd")
>>> val lengthGroup = words.groupBy { it.length }
>>> lengthGroup
{1=[a], 3=[abc, def], 2=[ab], 4=[abcd]}
groupBy(keySelector: (T) -> K, valueTransform: (T) -> V)
```

分组函数还有一个是 groupBy(keySelector: (T) -> K, valueTransform: (T) -> V)，根据条件选择器 keySelector 和转换函数 valueTransform 分组。代码示例：

```
>>> val programmer = listOf("K&R" to "C", "Bjar" to "C++", "Linus" to "C",
    "James" to "Java")
>>> programmer
[(K&R, C), (Bjar, C++), (Linus, C), (James, Java)]
>>> programmer.groupBy({it.second}, {it.first})
{C=[K&R, Linus], C++=[Bjar], Java=[James]}
```

这里涉及二元组 Pair 类，该类是 Kotlin 提供的用来处理二元数据组的。可以理解成 Map 中的一个键值对，比如 Pair（“key”，”value”）等价于“key” to “value”。

我们再通过下面的代码示例，来看一下这两个分组的区别：

```
>>> val words = listOf("a", "abc", "ab", "def", "abcd")
>>> words.groupBy( { it.length })
{1=[a], 3=[abc, def], 2=[ab], 4=[abcd]}
>>> words.groupBy( { it.length },{it.contains("b")})
{1=[false], 3=[true, false], 2=[true], 4=[true]}
```

我们可以看出，后者是在前者的基础上又映射了一次 {it.contains("b")}，把第 2 次映射的结果放到返回的 Map 中了。

2. groupingBy(crossinline keySelector: (T) -> K): Grouping<T, K>

另外，还可以使用 groupingBy(crossinline keySelector: (T) -> K): Grouping<T, K> 函数来创建一个 Grouping，然后调用计数函数 eachCount 统计分组：代码示例：

```
>>> val words = "one two three four five six seven eight nine ten".split(' ')
>>> words.groupingBy({it.first()}).eachCount()
{o=1, t=3, f=2, s=2, e=1, n=1}
```

上面的例子是统计 words 列表的元素单词中首字母出现的频数。

其中，eachCount 函数定义如下：

```
@SinceKotlin("1.1")
@JvmVersion
public fun <T, K> Grouping<T, K>.eachCount(): Map<K, Int> =
    // fold(0) { acc, e -> acc + 1 } optimized for boxing
    foldTo( destination = mutableMapOf(),
            initialValueSelector = { _, _ -> kotlin.jvm.internal.Ref.IntRef() },
            operation = { _, acc, _ -> acc.apply { element += 1 } })
    .mapValuesInPlace { it.value.element }
```

5.3.9 排序操作符

1. reversed(): List<T>

倒序排列集合元素。代码示例：

```
>>> val list = listOf(1,2,3)
>>> list.reversed()
[3, 2, 1]
```

这个函数，Kotlin 是直接调用的 java.util.Collections.reverse() 方法。其相关代码如下：

```
public fun <T> Iterable<T>.reversed(): List<T> {
    if (this is Collection && size <= 1) return toList()
    val list = toMutableList()
    list.reverse()
    return list
}

public fun <T> MutableList<T>.reverse(): Unit {
    java.util.Collections.reverse(this)
}
```

2. sorted 和 sortedDescending

升序排序和降序排序。代码示例：

```
>>> val list = listOf(1,3,2)
>>> list.sorted()
[1, 2, 3]
>>> list.sortedDescending()
[3, 2, 1]
```

3. sortedBy 和 sortedByDescending

可变集合 MutableList 的排序操作，根据函数映射的结果进行升序排序和降序排序。这两个函数定义如下：

```
public inline fun <T, R : Comparable<R>> MutableList<T>.sortBy(crossinline
    selector: (T) -> R?): Unit {
```

```
    if (size > 1) sortWith(compareBy(selector))
}
public inline fun <T, R : Comparable<R>> MutableList<T>.sortByDescending
    (crossinline selector: (T) -> R?): Unit {
    if (size > 1) sortWith(compareByDescending(selector))
}
```

代码示例：

```
>>> val mlist = mutableListOf("abc","c","bn","opqde","")
>>> mlist.sortBy({it.length})
>>> mlist
[, c, bn, abc, opqde]
>>> mlist.sortByDescending({it.length})
>>> mlist
[opqde, abc, bn, c, ]
```

5.3.10 生产操作符

常用的生产操作符如表 5-6 所示。

表 5-6 常用的生产操作符

函 数	功能说明
zip(other: Iterable<R>): List<Pair<T, R>>	两个集合按照下标配对，组合成的每个 Pair 作为新的 List 集合中的元素
partition(predicate: (T) -> Boolean): Pair<List<T>, List<T>>	根据条件将集合拆分成两个子集合组成的 Pair
plus(elements: Iterable<T>): List<T>	两个集合相加
plusElement(element: T): List<T>	向集合中添加一个元素

1. zip(other: Iterable<R>): List<Pair<T, R>>

两个集合按照下标配对，组合成的每个 Pair 作为新的 List 集合中的元素，并返回。如果两个集合长度不一样，取短的长度。代码示例：

```
>>> val list1 = listOf(1,2,3)
>>> val list2 = listOf(4,5,6,7)
>>> val list3 = listOf("x","y","z")
>>> list1.zip(list3)
[(1, x), (2, y), (3, z)]
>>> list3.zip(list1)
[(x, 1), (y, 2), (z, 3)]
>>> list2.zip(list3)
[(4, x), (5, y), (6, z)]  //取短的长度
>>> list3.zip(list2)
[(x, 4), (y, 5), (z, 6)]
>>> list1.zip(listOf<Int>())
[]
```

这个 zip 函数的定义如下：

```
public infix fun <T, R> Iterable<T>.zip(other: Iterable<R>): List<Pair<T, R>> {
    return zip(other) { t1, t2 -> t1 to t2 }
}
```

我们可以看出，其内部是调用了 zip(other) {t1, t2 -> t1 to t2}。这个函数定义如下：

```
public inline fun <T, R, V> Iterable<T>.zip(other: Iterable<R>, transform: (a:
    T, b: R) -> V): List<V> {
    val first = iterator()
    val second = other.iterator()
    val list = ArrayList<V>(minOf(collectionSizeOrDefault(10), other.
        collectionSizeOrDefault(10)))
    while (first.hasNext() && second.hasNext()) {
        list.add(transform(first.next(), second.next()))
    }
    return list
}
```

依次取两个集合相同索引的元素，使用提供的转换函数 transform 得到映射之后的值，作为元素组成一个新的 List，并返回该 List。列表的长度取两个集合中最短的。代码示例：

```
>>> val list1 = listOf(1,2,3)
>>> val list2 = listOf(4,5,6,7)
>>> val list3 = listOf("x","y","z")
>>> list1.zip(list3, {t1,t2 -> t2+t1})
[x1, y2, z3]
>>> list1.zip(list2, {t1,t2 -> t1*t2})
[4, 10, 18]
unzip(): Pair<List<T>, List<R>>
```

首先这个函数作用在元素是 Pair 的集合类上。依次取各个 Pair 元素的 first, second 值，分别放到 List、List 中，然后返回一个 first 为 List，second 为 List 的大的 Pair。函数定义：

```
public fun <T, R> Iterable<Pair<T, R>>.unzip(): Pair<List<T>, List<R>> {
    val expectedSize = collectionSizeOrDefault(10)
    val listT = ArrayList<T>(expectedSize)
    val listR = ArrayList<R>(expectedSize)
    for (pair in this) {
        listT.add(pair.first)
        listR.add(pair.second)
    }
    return listT to listR
}
```

看到这里，仍然有点抽象，我们直接看代码示例：

```
>>> val listPair = listOf(Pair(1,2),Pair(3,4),Pair(5,6))
```

```
>>> listPair
[(1, 2), (3, 4), (5, 6)]
>>> listPair.unzip()
([1, 3, 5], [2, 4, 6])
```

2. partition(predicate: (T) -> Boolean): Pair<List<T>, List<T>>

根据判断条件是否成立，将集合拆分成两个子集合组成的 Pair。我们可以直接看函数的定义来更加清晰的理解这个函数的功能：

```
public inline fun <T> Iterable<T>.partition(predicate: (T) -> Boolean): Pair<List<T>,
    List<T>> {
    val first = ArrayList<T>()
    val second = ArrayList<T>()
    for (element in this) {
        if (predicate(element)) {
            first.add(element)
        } else {
            second.add(element)
        }
    }
    return Pair(first, second)
}
```

我们可以看出，这是一个内联函数。代码示例：

```
>>> val list = listOf(1,2,3,4,5,6,7,8,9)
>>> list
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list.partition({it>5})
([6, 7, 8, 9], [1, 2, 3, 4, 5])
```

3. plus(elements: Iterable<T>): List<T>

合并两个 List。函数定义：

```
public operator fun <T> Iterable<T>.plus(elements: Iterable<T>): List<T> {
    if (this is Collection) return this.plus(elements)
    val result = ArrayList<T>()
    result.addAll(this)
    result.addAll(elements)
    return result
}
```

我们可以看出，这是一个操作符函数。可以用”+”替代。代码示例：

```
>>> val list1 = listOf(1,2,3)
>>> val list2 = listOf(4,5)
>>> list1.plus(list2)
[1, 2, 3, 4, 5]
>>> list1+list2
[1, 2, 3, 4, 5]
```

关于 plus 函数还有以下的重载函数：

- plus(element: T): List<T>
- plus(elements: Array<out T>): List<T>
- plus(elements: Sequence<T>): List<T>

4. plusElement(element: T): List<T>

在集合中添加一个元素。函数定义：

```
@kotlin.internal.InlineOnly
public inline fun <T> Iterable<T>.plusElement(element: T): List<T> {
    return plus(element)
}
```

我们可以看出，这个函数内部是直接调用的 plus(element: T): List<T>。代码示例：

```
>>> list1 + 10
[1, 2, 3, 10]
>>> list1.plusElement(10)
[1, 2, 3, 10]
>>> list1.plus(10)
[1, 2, 3, 10]
```

5.4 Set

类似的，Kotlin 中的 Set 也分为：不可变 Set 和可变 MutableSet（支持增加和删除）。

不可变 Set 同样是继承了 Collection。MutableSet 接口继承于 Set, MutableCollection，同时对 Set 进行扩展，添加了对元素添加和删除等操作。

Set 的类图结构如图 5-2 所示。

5.4.1 空集

万物生于无。我们先来看下 Kotlin 中的空集：

```
internal object EmptySet : Set<Nothing>, Serializable {
    private const val serialVersionUID: Long = 3406603774387020532

    override fun equals(other: Any?): Boolean = other is Set<*> && other.isEmpty()
    override fun hashCode(): Int = 0
    override fun toString(): String = "[]"

    override val size: Int get() = 0
    override fun isEmpty(): Boolean = true
    override fun contains(element: Nothing): Boolean = false
    override fun containsAll(elements: Collection<Nothing>): Boolean = elements.
        isEmpty()
```

```
    override fun iterator(): Iterator<Nothing> = EmptyIterator

    private fun readResolve(): Any = EmptySet
}
```

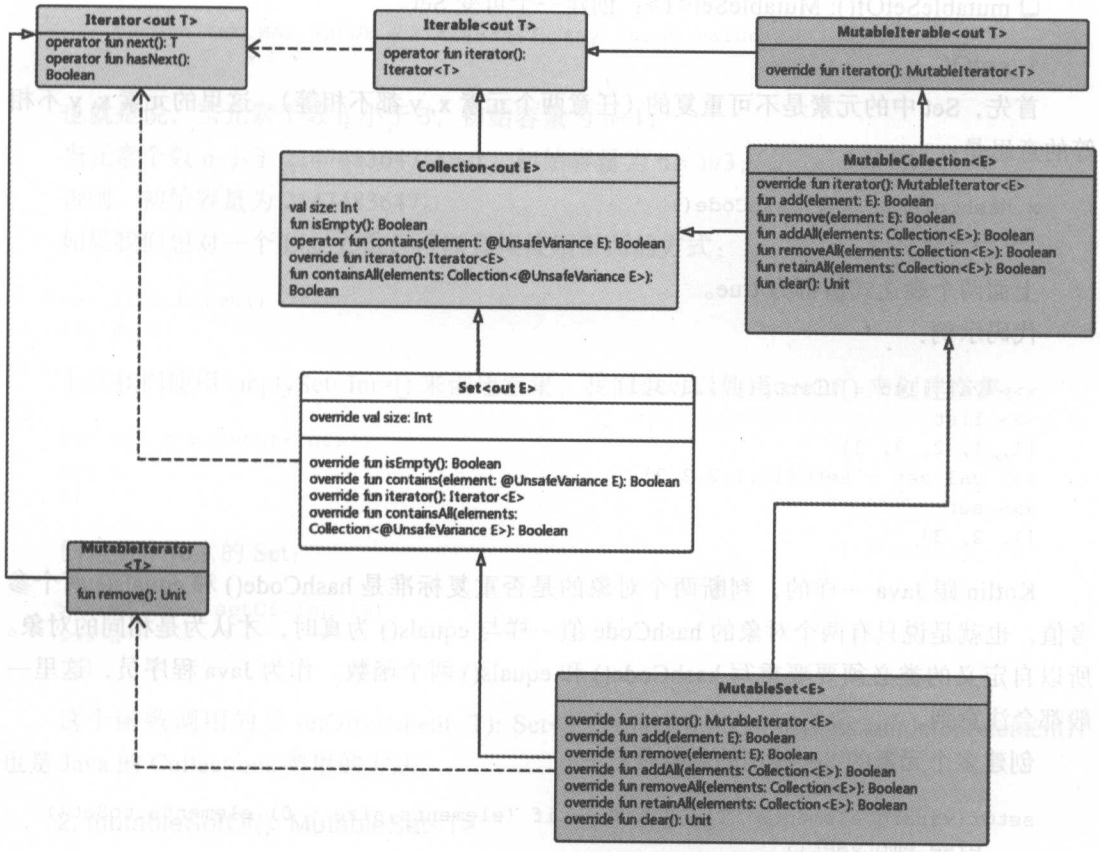


图 5-2 · Set 类图结构

空集继承了 Serializable，表明是可被序列化的。它的 size 是 0，isEmpty() 返回 true，hashCode() 也是 0。

下面是创建一个空集的代码示例：

```
>>> val emptySet = emptySet<Int>()
>>> emptySet
[]
>>> emptySet.size
0
>>> emptySet.isEmpty()
true
>>> emptySet.hashCode()
0
```

5.4.2 创建 Set

常用的创建 Set 的函数如下所示：

- setOf(vararg elements: T): Set<T>：使用元素 elements 创建一个 Set。
- mutableSetOf(): MutableSet<T>：创建一个可变 Set。

1. setOf

首先，Set 中的元素是不可重复的（任意两个元素 x, y 都不相等）。这里的元素 x, y 不相等的意思是：

```
x.hashCode() != y.hashCode()
!x.equals(y)
```

上面两个表达式值都为 true。

代码示例：

```
>>> val list = listOf(1,1,2,3,3)
>>> list
[1, 1, 2, 3, 3]
>>> val set = setOf(1,1,2,3,3)
>>> set
[1, 2, 3]
```

Kotlin 跟 Java 一样的，判断两个对象的是否重复标准是 hashCode() 和 equals() 两个参考值，也就是说只有两个对象的 hashCode 值一样与 equals() 为真时，才认为是相同的对象。所以自定义的类必须要要重写 hashCode() 和 equals() 两个函数。作为 Java 程序员，这里一般都会注意到。

创建多个元素的 Set 使用的函数是：

```
setOf(vararg elements: T): Set<T> = if (elements.size > 0) elements.toSet()
    else emptySet()
```

这个 toSet() 函数是 Array 类的扩展函数，定义如下：

```
public fun <T> Array<out T>.toSet(): Set<T> {
    return when (size) {
        0 -> emptySet()
        1 -> setOf(this[0])
        else -> toCollection(LinkedHashSet<T>(mapCapacity(size)))
    }
}
```

我们可以看出，setOf 函数背后实际上用的是 LinkedHashSet 构造函数。关于创建 Set 的初始容量的算法是：

```
@PublishedApi
internal fun mapCapacity(expectedSize: Int): Int {
```

```
    if (expectedSize < 3) {
        return expectedSize + 1
    }
    if (expectedSize < INT_MAX_POWER_OF_TWO) {
        return expectedSize + expectedSize / 3
    }
    return Int.MAX_VALUE // 2147483647, any large value
}
```

也就是说，当元素个数 n 小于 3，初始容量为 n+1；
当元素个数 n 小于 2147483647/2 + 1，初始容量为 n + n/3；
否则，初始容量为 2147483647。
如果我们想对一个 List 去重，可以直接使用下面的方式：

```
>>> list.toSet()
[1, 2, 3]
```

上文我们使用 emptySet<Int>() 来创建空集，我们也可以使用 setOf() 来创建空集：

```
>>> val s = setOf<Int>()
>>> s
[]
```

创建 1 个元素的 Set：

```
>>> val s = setOf<Int>(1)
>>> s
[1]
```

这个函数调用的是 setOf(element: T): Set<T> = java.util.Collections.singleton(element)，也是 Java 的 Collections 类里的方法。

2. mutableSetOf(): MutableSet<T>

创建一个可变 Set。
函数定义：

```
@SinceKotlin("1.1")
@kotlin.internal.InlineOnly
public inline fun <T> mutableSetOf(): MutableSet<T> = LinkedHashSet()
```

这个 LinkedHashSet() 构造函数背后实际上是 java.util.LinkedHashSet<E>，这就是 Kotlin 中的类型别名。

5.4.3 使用 Java 中的 Set 类

包 kotlin.collections 下面的 TypeAliases.kt 类中，有一些类型别名的定义如下：

```
@file:kotlin.jvm.JvmVersion
```

```
package kotlin.collections

@SinceKotlin("1.1") public typealias RandomAccess = java.util.RandomAccess


@SinceKotlin("1.1") public typealias ArrayList<E> = java.util.ArrayList<E>
@SinceKotlin("1.1") public typealias LinkedHashMap<K, V> = java.util.LinkedHashMap<K, V>
@SinceKotlin("1.1") public typealias HashMap<K, V> = java.util.HashMap<K, V>
@SinceKotlin("1.1") public typealias LinkedHashSet<E> = java.util.LinkedHashSet<E>
@SinceKotlin("1.1") public typealias HashSet<E> = java.util.HashSet<E>


//also @SinceKotlin("1.1")
internal typealias SortedSet<E> = java.util.SortedSet<E>
internal typealias TreeSet<E> = java.util.TreeSet<E>
```

从这里，我们可以看出，Kotlin 中的 LinkedHashSet、HashSet、SortedSet、TreeSet 就是直接使用的 Java 中的对应的集合类。

对应的创建的方法是：

- ❑ hashSetOf
- ❑ linkedSetOf
- ❑ mutableSetOf
- ❑ sortedSetOf

代码示例如下：

```
>>> val hs = hashSetOf(1,3,2,7)
>>> hs
[1, 2, 3, 7]
>>> hs::class
class java.util.HashSet
>>> val ls = linkedSetOf(1,3,2,7)
>>> ls
[1, 3, 2, 7]
>>> ls::class
class java.util.LinkedHashSet
>>> val ms = mutableSetOf(1,3,2,7)
>>> ms
[1, 3, 2, 7]
>>> ms::class
class java.util.LinkedHashSet
>>> val ss = sortedSetOf(1,3,2,7)
>>> ss
[1, 2, 3, 7]
>>> ss::class
class java.util.TreeSet
```

我们知道在 Java 中，Set 接口有两个主要的实现类 HashSet 和 TreeSet：

- HashSet：该类按照哈希算法来存取集合中的对象，存取速度较快。
- TreeSet：该类实现了 SortedSet 接口，能够对集合中的对象进行排序。
- LinkedHashSet：具有 HashSet 的查询速度，且内部使用链表维护元素的顺序，在对 Set 元素进行频繁插入、删除的场景中使用。

Kotlin 并没有单独去实现一套 HashSet、TreeSet 和 LinkedHashSet。如果我们在实际开发过程中，需要用到这些 Set，就可以直接用上面的方法。

5.4.4 Set 元素的加减操作 plus minus

Kotlin 中针对 Set 做了一些加减运算的扩展函数，例如：

```
operator fun <T> Set<T>.plus(element: T)
plusElement(element: T)
plus(elements: Iterable<T>)
operator fun <T> Set<T>.minus(element: T)
minusElement(element: T)
minus(elements: Iterable<T>)
```

代码示例：

```
>>> val ms = mutableSetOf(1,3,2,7)
>>> ms+10
[1, 3, 2, 7, 10]
>>> ms-1
[3, 2, 7]
>>>
>>> ms + listOf(8,9)
[1, 3, 2, 7, 8, 9]
>>> ms - listOf(8,9)
[1, 3, 2, 7]
>>> ms - listOf(1,3)
[2, 7]
```

5.5 Map

Map 是一种把键对象 Key 和值对象 Value 映射的集合，它的每一个元素都包含一对键对象和值对象（K-V Pair）。Key 可以看成是 Value 的索引，作为 key 的对象在集合中不可重复（uniq）。

如果我们从数据结构的本质上来看，其实 List 就是 Key 是 Int 类型下标的特殊的 Map。而 Set 也是 Key 为 Int，但是 Value 值不能重复的特殊 Map。

Kotlin 中的 Map 与 List、Set 一样，Map 也分为只读 Map 和可变的 MutableMap。

Map 没有继承于 Collection 接口，其类图结构如图 5-3 所示。

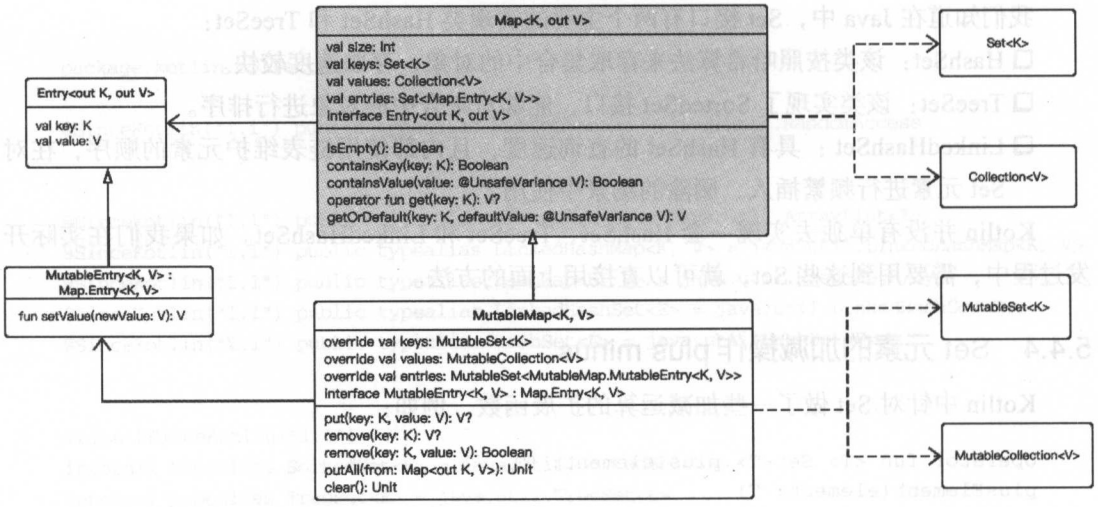


图 5-3 Map 类图结构

在接口 interface Map<K, out V> 中，K 是键值的类型，V 是对应的映射值的类型。这里的 out V 表示类型为 V 或 V 的子类。这是泛型的相关知识，我们将在下一章节中介绍。

其中，Entry<out K, out V> 中保存的是 Map 的键值对。

5.5.1 创建 Map

跟 Java 相比不同的是，在 Kotlin 中的 Map 区分了只读的 Map 和可编辑的 Map（MutableMap、HashMap、LinkedHashMap）。Kotlin 没有自己重新去实现一套集合类，而是在 Java 的集合类基础上做了一些扩展。

我们知道在 Java 中，根据内部数据结构的不同，Map 接口通常有多种实现类。其中常用的有：

- HashMap。HashMap 是基于哈希表（hash table）的 Map 接口的实现，以 key-value 的形式存在。在 HashMap 中，key-value 是一个整体，系统会根据 hash 算法来来计算 key-value 的存储位置，我们可以通过 key 快速地存取 value。它允许使用 null 值和 null 键。另外，HashMap 中元素的顺序，随着时间的推移会发生变化。
- TreeMap。使用红黑二叉树（red-black tree）的 Map 接口的实现。
- LinkedHashMap。还有继承了 HashMap，并使用链表实现的 LinkedHashMap。LinkedHashMap 保存了记录的插入顺序，在用 Iterator 遍历 LinkedHashMap 时，先得到的记录是先插入的记录。简单说，LinkedHashMap 是有序的，它使用链表维护内部次序。

我们在使用 Kotlin 创建 Map 的时候，实际上大部分都是调用 Java 的 Map 的方法。

下面我们就来介绍 Map 的创建以及基本操作函数。

Map 的创建函数如表 5-7 所示。

表 5-7 创建 Map 的函数及功能说明

函数名	功能功能
mapOf()	创建一个只读空 Map
mapOf(pair: Pair<K, V>): Map<K, V>	使用二元组 Pair 创建一个只读 Map
mutableMapOf()	创建一个空的可变的 Map
mutableMapOf(vararg pairs: Pair<K, V>): MutableMap<K, V>	创建一个可编辑的 MutableMap 对象
hashMapOf(): HashMap<K, V>	创建 HashMap 对象
linkedMapOf(): LinkedHashMap<K, V>	创建 LinkedHashMap 对象
linkedMapOf(vararg pairs: Pair<K, V>): LinkedHashMap<K, V>	创建带二元组 Pair 元素的 LinkedHashMap 对象
sortedMapOf(vararg pairs: Pair<K, V>): SortedMap<K, V>	创建一个根据 Key 升序排序好的 TreeMap

1. mapOf()

创建一个只读空 Map。示例如下：

```
>>> val map1 = mapOf<String, Int>()
>>> map1.size
0
>>> map1.isEmpty()
true
```

我们还可以用另外一个函数创建空 Map：

```
>>> val map2 = emptyMap<String, Int>()
>>> map2.size
0
>>> map2.isEmpty()
true
```

空 Map 都是相等的：

```
>>> map2==map1
true
```

这个空 Map 是只读的，其属性和函数返回都是预定义好的，其代码如下：

```
private object EmptyMap : Map<Any?, Nothing>, Serializable {
    private const val serialVersionUID: Long = 8246714829545688274

    override fun equals(other: Any?): Boolean = other is Map<*,*> && other.isEmpty()
    override fun hashCode(): Int = 0
    override fun toString(): String = "{}"

    override val size: Int get() = 0
```

```
    override fun isEmpty(): Boolean = true

    override fun containsKey(key: Any?): Boolean = false
    override fun containsValue(value: Nothing): Boolean = false
    override fun get(key: Any?): Nothing? = null
    override val entries: Set<Map.Entry<Any?, Nothing>> get() = EmptySet
    override val keys: Set<Any?> get() = EmptySet
    override val values: Collection<Nothing> get() = EmptyList

    private fun readResolve(): Any = EmptyMap
}
```

2. mapOf(pair: Pair<K, V>): Map<K, V>

使用二元组 Pair 创建一个只读 Map，示例如下：：

```
>>> val map = mapOf(1 to "x", 2 to "y", 3 to "z")
>>> map
{1=x, 2=y, 3=z}
>>> map.get(1)
x
>>> map.get(3)
z
>>> map.size
3
>>> map.entries
[1=x, 2=y, 3=z]
```

这个创建函数内部是调用的 LinkedHashMap 构造函数，其相关代码如下：

```
pairs.toMap(LinkedHashMap(mapCapacity(pairs.size)))
```

如果我们想编辑这个 Map，编译器会直接报错：

```
>>> map[1]="a"
error: unresolved reference. None of the following candidates is applicable
    because of receiver type mismatch:
@InlineOnly public operator inline fun <K, V> MutableMap<Int, String>.set(key:
    Int, value: String): Unit defined in kotlin.collections
@InlineOnly public operator inline fun kotlin.text.StringBuilder /* = java.lang.
    StringBuilder */.set(index: Int, value: Char): Unit defined in kotlin.text
map[1]="a"
^
error: no set method providing array access
map[1]="a"
   ^
```

因为在不可变（Immutable）Map 中，根本就没有提供 set 函数。

3. mutableMapOf()

创建一个空的可变的 Map，示例如下：

```
>>> val map = mutableMapOf<Int, Any?>()
>>> map.isEmpty()
true
>>> map[1] = "x"
>>> map[2] = 1
>>> map
{1=x, 2=1}
```

该函数直接是调用的 LinkedHashMap() 构造函数。

4. mutableMapOf(vararg pairs: Pair<K, V>): MutableMap<K, V>

创建一个可编辑的 MutableMap 对象，示例如下：

```
>>> val map = mutableMapOf(1 to "x", 2 to "y", 3 to "z")
>>> map
{1=x, 2=y, 3=z}
>>> map[1]="a"
>>> map
{1=a, 2=y, 3=z}
```

另外，如果 Map 中有重复的 key 键，后面的会直接覆盖掉前面的：

```
>>> val map = mutableMapOf(1 to "x", 2 to "y", 1 to "z")
>>> map
{1=z, 2=y}
```

后面的 1 to "z" 直接把前面的 1 to "x" 覆盖掉了。

5. hashMapOf(): HashMap<K, V>

创建 HashMap 对象。Kotlin 直接使用的是 Java 的 HashMap，示例如下：

```
>>> val map: HashMap<Int, String> = hashMapOf(1 to "x", 2 to "y", 3 to "z")
>>> map
{1=x, 2=y, 3=z}
```

6. linkedMapOf(): LinkedHashMap<K, V>

创建空对象 LinkedHashMap。直接使用的是 Java 中的 LinkedHashMap，示例如下：

```
>>> val map: LinkedHashMap<Int, String> = linkedMapOf()
>>> map
{}
>>> map[1]="x"
>>> map
{1=x}
```

7. linkedMapOf(vararg pairs: Pair<K, V>): LinkedHashMap<K, V>

创建带二元组 Pair 元素的 LinkedHashMap 对象。直接使用的是 Java 中的 LinkedHashMap，示例如下：

```
>>> val map: LinkedHashMap<Int, String> = linkedMapOf(1 to "x", 2 to "y", 3 to "z")
```

```
>>> map
{1=x, 2=y, 3=z}
>>> map[1]="a"
>>> map
{1=a, 2=y, 3=z}
```

8. sortedMapOf(vararg pairs: Pair<K, V>): SortedMap<K, V>

创建一个根据 Key 升序排序好的 TreeMap。对应的是使用 Java 中的 SortedMap，示例如下：

```
>>> val map = sortedMapOf(Pair("c", 3), Pair("b", 2), Pair("d", 1))
>>> map
{b=2, c=3, d=1}
```

5.5.2 访问 Map 的元素

Map 的元素包含的属性有：entries、keys、values size，还有一个操作符函数 Map.get(key)，等同于使用 Map[Key]。下面我们分别介绍。

1. entries 属性

我们可以直接访问 entries 属性：

```
val entries: Set<Entry<K, V>>
```

获取该 Map 中的所有键 / 值对的 Set。这个 Entry 类型定义如下：

```
public interface Entry<out K, out V> {
        public val key: K
        public val value: V
    }
```

代码示例：

```
>>> val map = mapOf("x" to 1, "y" to 2, "z" to 3)
>>> map
{x=1, y=2, z=3}
>>> map.entries
[x=1, y=2, z=3]
```

这样，我们就可以遍历这个 Entry 的 Set 了：

```
>>> map.entries.forEach({println("key="+ it.key + " value=" + it.value)})
key=x value=1
key=y value=2
key=z value=3
```

2. keys 属性

访问 keys 属性：

```
val keys: Set<K>
```

获取 Map 中的所有键的 Set。

```
>>> map.keys
[x, y, z]
```

3. values 属性

访问 val values: Collection<V> 获取 Map 中的所有值的 Collection。这个值的集合可能包含重复值：

```
>>> map.values
[1, 2, 3]
```

4. size 属性

访问 val size: Int 获取 map 键 / 值对的数目。

```
>>> map.size
3
```

5. get(key: K)

我们使用 get 函数来通过 key 来获取 value 的值：

```
operator fun get(key: K): V?
```

对应的操作符是 []：

```
>>> map["x"]
1
>>> map.get("x")
1
```

如果这个 key 不在 Map 中，就返回 null：

```
>>> map["k"]
null
```

如果不想返回 null，可以使用 getOrDefault 函数：

```
getOrDefault(key: K, defaultValue: @UnsafeVariance V): V
```

当为 null 时，不返回 null，而是返回设置的一个默认值：

```
>>> map.getOrDefault("k",0)
0
```

这个默认值的类型，要和 V 对应。类型不匹配会报错：

```
>>> map.getOrDefault("k","a")
error: type mismatch: inferred type is String but Int was expected
map.getOrDefault("k","a")
                  ^
```

5.5.3 Map 操作符函数

Map 中的常用的操作符函数如表 5-8 所示。

表 5-8 Map 操作符函数及功能说明

函数名称	功能说明
containsKey(key: K): Boolean	是否包含该 key
containsValue(value: V): Boolean	是否包含该 value
component1() □component2()	分别用来直接访问 key 和 value
Map.Entry<K, V>.toPair(): Pair<K, V>	把 Map 的 Entry 转换为 Pair
getOrElse(key: K, defaultValue: () -> V): V	通过 key 获取值，当没有值可以设置默认值
getValue(key: K): V	当 Map 中不存在这个 key，调用 get 函数，如果不想返回 null，直接抛出异常
getOrPut(key: K, defaultValue: () -> V): V	如果不存在这个 key，就添加这个 key 到 Map 中，对应的 value 是 defaultValue
iterator(): Iterator<Map.Entry<K, V>>	这个函数返回的是 entries.iterator()。这样我们就使用 for 循环来遍历 Map
mapKeys(transform: (Map.Entry<K, V>) -> R): Map<R, V>	把 Map 的 Key 设置为通过转换函数 transform 映射之后的值
mapValues(transform: (Map.Entry<K, V>) -> R): Map<K, R>	把 Map 的 value 设置为通过转换函数 transform 转换之后的新值
filterKeys(predicate: (K) -> Boolean): Map<K, V>	返回过滤出满足 key 判断条件的元素组成的新 Map
filterValues(predicate: (V) -> Boolean): Map<K, V>	返回过滤出满足 value 判断条件的元素组成的新 Map
filter(predicate: (Map.Entry<K, V>) -> Boolean): Map<K, V>	返回过滤出满足 Entry 判断条件的元素组成的新 Map
Iterable<Pair<K, V>>.toMap(destination: M): M	把持有 Pair 的 Iterable 集合转换为 Map
Map<out K, V>.toMutableMap(): MutableMap<K, V>	把一个只读的 Map 转换为可编辑的 MutableMap
put(key: K, value: V): V?	根据 key 设置元素的 value。如果该 key 存在就更新 value；不存在就添加，但是 put 的返回值是 null
MutableMap<out K, V>.remove(key: K): V?	根据键值 key 来删除元素
MutableMap<K, V>.clear(): Unit	清空 MutableMap

1. containsKey(key: K): Boolean

是否包含该 key，示例如下：

```
>>> val map = mapOf("x" to 1, "y" to 2, "z" to 3)
>>> map.containsKey("x")
true
>>> map.containsKey("j")
false
```

2. containsValue(value: V): Boolean

是否包含该 value，示例如下：

```
>>> val map = mapOf("x" to 1, "y" to 2, "z" to 3)
>>> map.containsValue(2)
true
>>> map.containsValue(20)
false
```

3. component1() component2()

Map.Entry<K, V> 的操作符函数，分别用来直接访问 key 和 value，示例如下：

```
>>> val map = mapOf("x" to 1, "y" to 2, "z" to 3)
>>> map.entries.forEach({println("key="+ it.component1() + " value=" + it.
    component2())})
key=x value=1
key=y value=2
key=z value=3
```

这两个函数的定义如下：

```
@kotlin.internal.InlineOnly
public inline operator fun <K, V> Map.Entry<K, V>.component1(): K = key

@kotlin.internal.InlineOnly
public inline operator fun <K, V> Map.Entry<K, V>.component2(): V = value
```

4. Map. Entry<K, V>. toPair(): Pair<K, V>

把 Map 的 Entry 转换为 Pair，示例如下：

```
>>> map.entries
[x=1, y=2, z=3]
>>> map.entries.forEach({println(it.toPair())})
(x, 1)
(y, 2)
(z, 3)
```

5. getOrElse(key: K, defaultValue: () -> V): V

通过 key 获取值，当没有值可以设置默认值，示例如下：

```
>>> val map = mutableMapOf<String, Int?>()
>>> map.getOrElse("x", { 1 })
1
>>> map["x"] = 3
>>> map.getOrElse("x", { 1 })
3
```

6. getValue(key: K): V

当 Map 中不存在这个 key，调用 get 函数，如果不想返回 null，直接抛出异常，可调用

此方法：

```
val map = mutableMapOf<String, Int?>()
>>> map.get("v")
null
>>> map.getValue("v")
java.util.NoSuchElementException: Key v is missing in the map.
    at kotlin.collections.MapsKt__MapWithDefaultKt.getOrImplicitDefaultNullable
       (MapWithDefault.kt:19)
    at kotlin.collections.MapsKt__MapsKt.getValue(Maps.kt:252)
```

7. getOrPut(key: K, defaultValue: () -> V): V

如果不存在这个 key，就添加这个 key 到 Map 中，对应的 value 是 defaultValue：

```
>>> val map = mutableMapOf<String, Int?>()
>>> map.getOrPut("x", { 2 })
2
>>> map
{x=2}
```

8. iterator(): Iterator<Map. Entry<K, V>>

这个函数返回的是 entries.iterator()。这样我们就可以像下面这样使用 for 循环来遍历 Map：

```
>>> val map = mapOf("x" to 1, "y" to 2, "z" to 3 )
>>> for((k,v) in map){println("key=$k, value=$v")}
key=x, value=1
key=y, value=2
key=z, value=3
```

9. mapKeys(transform: (Map. Entry<K, V>) -> R): Map<R, V>

把 Map 的 Key 设置为通过转换函数 transform 映射之后的值，示例如下：

```
>>> val map:Map<Int,String> = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> val mmap = map.mapKeys{it.key * 10}
>>> mmap
{10=a, 20=b, 30=c, -10=z}
```

注意，这里的 it 是 Map 的 Entry。

如果不巧，有任意两个 key 通过映射之后相等了，那么后面的 key 将会覆盖掉前面的 key：

```
>>> val mmap = map.mapKeys{it.key * it.key}
>>> mmap
{1=z, 4=b, 9=c}
```

我们可以看出，1 to "a" 被 −1 to "z" 覆盖掉了。

10. mapValues(transform: (Map. Entry<K, V>) -> R): Map<K, R>

对应的这个函数是把 Map 的 value 设置为通过转换函数 transform 转换之后的新值，示例如下：

```
>>> val map:Map<Int,String> = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> val mmap = map.mapValues({it.value + "$"})
>>> mmap
{1=a$, 2=b$, 3=c$, -1=z$}
```

11. filterKeys(predicate: (K) -> Boolean): Map<K, V>

返回过滤出满足 key 判断条件的元素组成的新 Map，示例如下：

```
>>> val map:Map<Int,String> = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map.filterKeys({it>0})
{1=a, 2=b, 3=c}
```

注意，这里的 it 元素是 Key。

12. filterValues(predicate: (V) -> Boolean): Map<K, V>

返回过滤出满足 value 判断条件的元素组成的新 Map，示例如下：

```
>>> val map:Map<Int,String> = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map.filterValues({it>"b"})
{3=c, -1=z}
```

注意，这里的 it 元素是 value。

13. filter(predicate: (Map. Entry<K, V>) -> Boolean): Map<K, V>

返回过滤出满足 Entry 判断条件的元素组成的新 Map，示例如下：

```
>>> val map:Map<Int,String> = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map.filter({it.key>0 && it.value > "b"})
{3=c}
```

14. Iterable<Pair<K, V>>. toMap(destination: M): M

把持有 Pair 的 Iterable 集合转换为 Map，示例如下：

```
>>> val pairList = listOf(Pair(1,"a"),Pair(2,"b"),Pair(3,"c"))
>>> pairList
[(1, a), (2, b), (3, c)]
>>> pairList.toMap()
{1=a, 2=b, 3=c}
```

15. Map<out K, V>. toMutableMap(): MutableMap<K, V>

把一个只读的 Map 转换为可编辑的 MutableMap，示例如下：

```
>>> val map = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map[1]="x"
error: unresolved reference. None of the following candidates is applicable ...
error: no set method providing array access
map[1]="x"
 ^

>>> val mutableMap = map.toMutableMap()
```

```
>>> mutableMap
{1=a, 2=b, 3=c, -1=z}
>>> mutableMap[1]="x"
>>> mutableMap
{1=x, 2=b, 3=c, -1=z}
plus minus
```

Map 的加法运算符函数如下：

```
operator fun <K, V> Map<out K, V>.plus(pair: Pair<K, V>): Map<K, V>
operator fun <K, V> Map<out K, V>.plus(pairs: Iterable<Pair<K, V>>): Map<K, V>
operator fun <K, V> Map<out K, V>.plus(pairs: Array<out Pair<K, V>>): Map<K, V>
operator fun <K, V> Map<out K, V>.plus(pairs: Sequence<Pair<K, V>>): Map<K, V>
operator fun <K, V> Map<out K, V>.plus(map: Map<out K, V>): Map<K, V>
```

代码示例：

```
>>> val map = mapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map+Pair(10,"g")
{1=a, 2=b, 3=c, -1=z, 10=g}
>>> map + listOf(Pair(9,"s"),Pair(10,"w"))
{1=a, 2=b, 3=c, -1=z, 9=s, 10=w}
>>> map + arrayOf(Pair(9,"s"),Pair(10,"w"))
{1=a, 2=b, 3=c, -1=z, 9=s, 10=w}
>>> map + sequenceOf(Pair(9,"s"),Pair(10,"w"))
{1=a, 2=b, 3=c, -1=z, 9=s, 10=w}
>>> map + mapOf(9 to "s", 10 to "w")
{1=a, 2=b, 3=c, -1=z, 9=s, 10=w}
```

加并赋值函数：

```
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(pair: Pair<K, V>)
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(pairs: Iterable
    <Pair<K, V>>)
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(pairs: Array<out
    Pair<K, V>>)
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(pairs: Sequence
    <Pair<K, V>>)
inline operator fun <K, V> MutableMap<in K, in V>.plusAssign(map: Map<K, V>)
```

代码示例：

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map+=Pair(10,"g")
>>> map
{1=a, 2=b, 3=c, -1=z, 10=g}
>>> map += listOf(Pair(9,"s"),Pair(11,"w"))
>>> map
{1=a, 2=b, 3=c, -1=z, 10=g, 9=s, 11=w}
>>> map += mapOf(20 to "qq", 30 to "tt")
>>> map
{1=a, 2=b, 3=c, -1=z, 10=g, 9=s, 11=w, 20=qq, 30=tt}
```

减法跟加法类似。

16. put(key: K, value: V): V?

根据 key 设置元素的 value。如果该 key 存在就更新 value；不存在就添加，但是 put 的返回值是 null，示例如下：

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map
{1=a, 2=b, 3=c, -1=z}
>>> map.put(10,"q")
null
>>> map
{1=a, 2=b, 3=c, -1=z, 10=q}
>>> map.put(1,"f")
a
>>> map
{1=f, 2=b, 3=c, -1=z, 10=q}
putAll(from: Map<out K, V>): Unit
```

把一个 Map 全部添加到一个 MutableMap 中：

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> val map2 = mapOf(99 to "aa", 100 to "bb")
>>> map.putAll(map2)
>>> map
{1=a, 2=b, 3=c, -1=z, 99=aa, 100=bb}
```

如果有 key 重复的，后面的值会覆盖掉前面的值：

```
>>> map
{1=a, 2=b, 3=c, -1=z, 99=aa, 100=bb}
>>> map.putAll(mapOf(1 to "www",2 to "tttt"))
>>> map
{1=www, 2=tttt, 3=c, -1=z, 99=aa, 100=bb}
```

17. MutableMap<out K, V>. remove(key: K): V?

根据键值 key 来删除元素，示例如下：

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map.remove(-1)
z
>>> map
{1=a, 2=b, 3=c}
>>> map.remove(100)
null
>>> map
{1=a, 2=b, 3=c}
```

18. MutableMap<K, V>. clear(): Unit

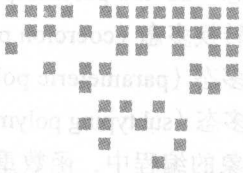
清空 MutableMap，示例如下：

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c", -1 to "z")
>>> map
{1=a, 2=b, 3=c, -1=z}
>>> map.clear()
>>> map
{}
```

5.6 本章小结

Kotlin 的集合类没有像 Scala 那样自己单独来实现一套，而是在兄弟语言 Java 集合类的基础上进行了扩展。Kotlin 的集合类中提供大量的函数式 API，这些丰富的扩展函数使得我们的代码写起来更加顺畅。同时，Kotlin 也汲取了 Scala 的集合类的优秀特性，比如引入了不可变集合类。

集合类持有的是对象，而怎样地放入正确的对象类型则是我们写代码过程中需要注意的。下一章节中我们将学习泛型。



第 6 章 Chapter 6

泛　型

通常情况的类和函数，我们只需要使用具体的类型即可：要么是基本类型，要么是自定义的类。

但是尤其在集合类的场景下，我们需要编写可以应用于多种类型的代码，最简单原始的做法是，针对每一种类型，写一套刻板的代码。这样做，代码复用率会很低，抽象也没有做好。另外，我们向集合类中存储什么样类型的对象，如果我们不做显式地限制，往往会出现类型强制转换安全等问题。

而泛型就可以解决这些问题。本章将介绍 Kotlin 泛型的基本概念、型变以及类型边界等内容，同时简单介绍了泛型类与泛型函数。

6.1　泛型简介

在 JDK 5 中，Java 引用了泛型。泛型（Generic Type）即“参数化类型”（Parameterized Type），顾名思义，就是将类型由原来的具体的类型改为参数，类似于方法中的变量参数，此时类型也定义成参数形式，我们称之为类型参数，然后在使用时传入具体的类型（类型实参）。

我们知道，在数学中泛函是以函数为自变量的函数。编程中的泛型就是以类型 T 为变量的类型，即参数化类型。这样的类型变量就叫类型参数（Type Parameters），简单讲就是将类型由原来的具体的类型“参数化”。

6.1.1　泛型与多态

完整的多态（polymorphic）概念由以下三个组成部分：

- 特殊多态（ad hoc polymorphism），此类下又包含函数重载（function overloading）以及类型转换多态（coercion polymorphism）。
- 参数化多态（parameteric polymorphism），对应到 Java 里即泛型。
- 子类型多态（subtyping polymorphism），即通过类型继承关系获得的多态。

在面向对象的编程中，函数重载、继承 / 方法重写、泛型、运算符重载等等都体现了多态。

6.1.2 为何引进泛型

我们先来看看没有泛型之前，我们的集合类是怎样持有对象的。

在 Java 中，Object 类是所有类的根类。为了集合类的通用性。我们把元素的类型定义为 Object，当放入具体的类型时，再进行强制类型转换。

下面一个示例代码：

```
class RawArrayList {
    public int length = 0;
    private Object[] elements;

    public RawArrayList(int length) {
        this.length = length;
        this.elements = new Object[length];
    }

    public Object get(int index) {
        return elements[index];
    }

    public void add(int index, Object element) {
        elements[index] = element;
    }

    @Override
    public String toString() {
        return "RawArrayList{" +
            "length=" + length +
            ", elements=" + Arrays.toString(elements) +
            '}';
    }
}
```

一个简单的测试代码如下：

```
public class RawTypeDemo {

    public static void main(String[] args) {
        RawArrayList rawArrayList = new RawArrayList(4);
```

```
        rawArrayList.add(0, "a");
        rawArrayList.add(1, "b");
        System.out.println(rawArrayList);

        String a = (String)rawArrayList.get(0);
        System.out.println(a);

        String b = (String)rawArrayList.get(1);
        System.out.println(b);

        rawArrayList.add(2, 200);
        rawArrayList.add(3, 300);
        System.out.println(rawArrayList);

        int c = (int)rawArrayList.get(2);
        int d = (int)rawArrayList.get(3);
        System.out.println(c);
        System.out.println(d);

        //强制转型异常: Exception in thread "main" java.lang.ClassCastException:
        //java.lang.Integer cannot be cast to java.lang.String
        String x = (String)rawArrayList.get(2);
        System.out.println(x);

    }

}
```

可以看出，在使用原生态类型实现的集合类中，我们使用的是 Object[] 数组。这种实现方式存在以下问题：

- 向集合中添加对象元素的时候，没有对元素的类型进行检查，也就是说，我们向集合中添加任意对象，编译器都不会报错。
- 当我们从集合中获取一个值的时候，不能都使用 Object 类型，需要进行强制类型转换。而这个转换过程由于在添加元素的时候没有作任何的类型限制与检查，所以容易出错。例如上面代码中的：

```
String a = (String)rawArrayList.get(0);
```

对于这行代码，编译时不会报错，但是运行时会抛出类型转换错误。

6.1.3 Java 与 Kotlin 中的泛型

由于我们不能笼统地把集合类中所有的对象视作 Object，然后在使用的时候各自作强制类型转换。因此，我们引入了类型参数来解决这个类型安全使用的问题。

Java 中的泛型是在 1.5 版本之后加入的，我们可以为类和方法分别定义泛型参数，比如 Java 中的 Map 接口的定义：

```
public interface Map<K,V> {
    ...
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);
    V put(K key, V value);
    V remove(Object key);
    void putAll(Map<? extends K, ? extends V> m);
    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K, V>> entrySet();
    default V getOrDefault(Object key, V defaultValue) {
        V v;
        return (((v = get(key)) != null) || containsKey(key))
            ? v
            : defaultValue;
    }
}
```

我们在 Kotlin 中的写法基本一样：

```
public interface Map<K, out V> {
    ...
    public fun containsKey(key: K): Boolean
    public fun containsValue(value: @UnsafeVariance V): Boolean
    public operator fun get(key: K): V?
    @SinceKotlin("1.1")
    @PlatformDependent
    public fun getOrDefault(key: K, defaultValue: @UnsafeVariance V): V {
        // See default implementation in JDK sources
        return null as V
    }
    public val keys: Set<K>
    public val values: Collection<V>
    public val entries: Set<Map.Entry<K, V>>
}
public interface MutableMap<K, V> : Map<K, V> {
    public fun put(key: K, value: V): V?
    public fun remove(key: K): V?
    public fun putAll(from: Map<out K, V>): Unit
    ...

}
```

比如，在实例化一个 Map 时，我们使用如下函数：

```
fun <K, V> mapOf(vararg pairs: Pair<K, V>): Map<K, V>
```

类型参数 K、V 是一个占位符，当泛型类型被实例化和使用时，它将被一个实际的类型参数所替代。

代码示例：

```
>>> val map = mutableMapOf<Int,String>(1 to "a", 2 to "b", 3 to "c")
>>> map
{1=a, 2=b, 3=c}
>>> map.put(4,"c")
null
>>> map
{1=a, 2=b, 3=c, 4=c}
```

mutableMapOf<Int, String> 表示参数化类型 <K, V> 分别是 Int 和 String，这是泛型类型集合的实例化。在这里，放置 K、V 的位置被具体的 Int 和 String 类型所替代。

泛型主要是用来限制集合类持有的对象类型，这样使得类型更加安全。当我们在一个集合类里面放入了错误类型的对象，编译器就会报错：

```
>>> map.put("5","e")
error: type mismatch: inferred type is String but Int was expected
map.put("5","e")
        ^
```

Kotlin 中有类型推断的功能，有些类型参数可以直接省略不写。上面的 mapOf 后面的类型参数可以省掉不写：

```
>>> val map = mutableMapOf(1 to "a", 2 to "b", 3 to "c")
>>> map
{1=a, 2=b, 3=c}
```

Java 和 Kotlin 的泛型实现，都采用了运行时类型擦除的方式。也就是说，在运行时，这些类型参数的信息将会被擦除。Java 和 Kotlin 的泛型对于语法的约束是在编译期进行的。

6.2 型变

型变（Variant）是一个令人费解的概念，但却是理解类型系统的重要基石，它有三种基本形态：协变（Covariant）、逆变（Contravariant）、不变（Invariant）。下面我们就从 Java 中的类型通配符开始介绍这三个概念。

6.2.1 Java 的类型通配符

Java 泛型的通配符有以下两种形式：

- ? extends T：子类型上界限定符，指定类型参数的上限（该类型必须是类型 T 或者它的子类型）。
- ? super T：超类型下界限定符，指定类型参数的下限（该类型必须是类型 T 或者它的父类型）。

我们称之为类型通配符（Type Wildcard）。

代码示例：

```
class Animal {

    public void act(List<? extends Animal> list) {
        for (Animal animal : list) {
            animal.eat();
        }
    }

    public void aboutShepherdDog(List<? super ShepherdDog> list) {
        System.out.println("About ShepherdDog");
    }

    public void eat() {
        System.out.println("Eating");
    }

}

class Dog extends Animal {}

class Cat extends Animal {}

class ShepherdDog extends Dog {}
```

在方法 act(List<? extends Animal> list) 中，list 可以传入以下类型的参数：

```
List<Animal>
List<Dog>
List<ShepherdDog>
List<Cat>
```

测试代码：

```
List<Animal> list3 = new ArrayList<>();
list3.add(new Dog());
list3.add(new Cat());
animal.act(list3);

List<Dog> list4 = new ArrayList<>();
list4.add(new Dog());
list4.add(new Dog());
animal.act(list4);

List<Cat> list5 = new ArrayList<>();
list5.add(new Cat());
list5.add(new Cat());
animal.act(list5);
```

为了更加简单明了说明这些类型的层次关系，如图 6-1 和图 6-2 所示。

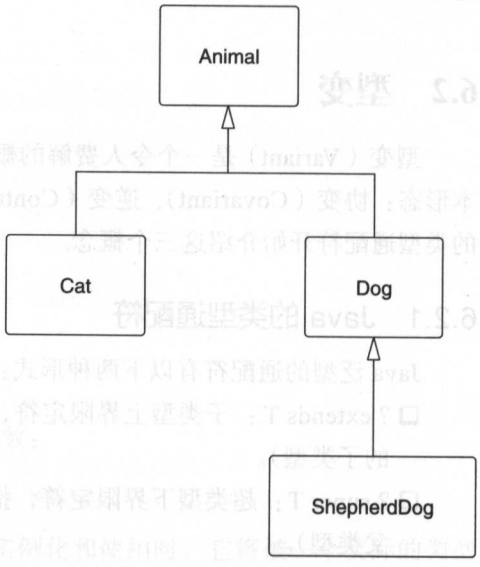


图 6-1 对象层次类图

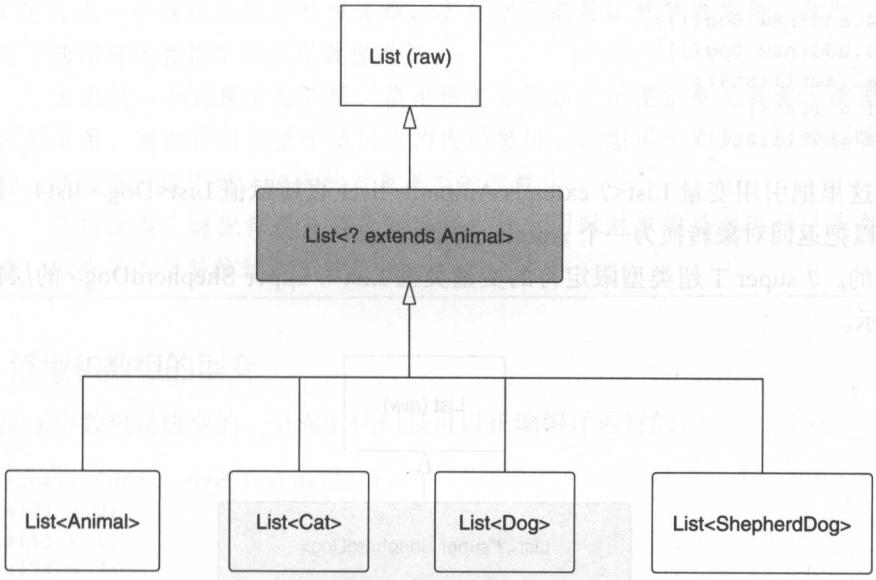


图 6-2 集合类泛型层次类图

也就是说，List 并不是 List 的子类型，而是两种不存在父子关系的类型。

而 List<? extends Animal> 是 List<Animal>、List<Dog> 等的父类型，对于任何的 List<X>，这里的 X 只要是 Animal 的子类型，那么 List<? extends Animal> 就是 List<X> 的父类型。

使用通配符 List<? extends Animal> 的引用，我们不可以往这个 List 中添加 Animal 类型及其子类型的元素，如下所示：

```
List<? extends Animal> list1 = new ArrayList<>();

list1.add(new Dog());
list1.add(new Animal());
```

这样的写法，Java 编译器是不允许的：

```
17
18      List<? extends Animal> list1 = new ArrayList<>();
19
20      list1.add(new Dog());
21      list1.add(new Animal());
```

add (capture<? extends com.easy.kotlin.Animal>) In List cannot be applied

to (com.easy.kotlin.Animal)

因为对于 set 方法，编译器无法知道具体的类型，所以会拒绝这个调用。但是，如果是 get 方法形式的调用，则是允许的，如下所示：

```
List<? extends Animal> list1 = new ArrayList<>();
List<Dog> list4 = new ArrayList<>();
```

```
list4.add(new Dog());
list4.add(new Dog());
animal.act(list4);
list1 = list4;
animal.act(list1);
```

我们这里把引用变量 List<? extends Animal> list1 直接赋值 List<Dog> list4，因为编译器知道可以把返回对象转换为一个 Animal 类型。

相应的，? super T 超类型限定符的变量类型 List<? super ShepherdDog> 的层次结构如图 6-3 所示。

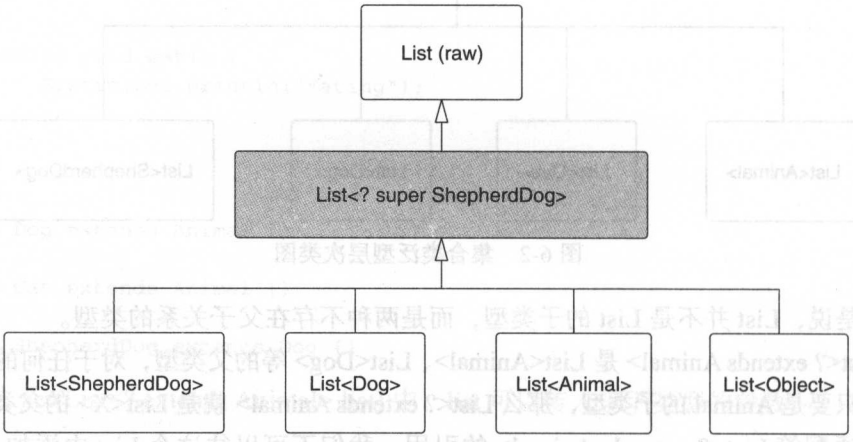


图 6-3 List<? super ShepherdDog> 的层次结构

在 Java 中，还有一个无界通配符，即单独一个 ?。如 List<?>，? 可以代表任意类型，“任意”是未知类型。例如：

```
Pair<?>
```

参数替换后的 Pair 类有如下方法：

```
? getFirst()
void setFirst(?)
```

我们可以调用 getFirst 方法，因为编译器可以把返回值转换为 Object。

但是不能调用 setFirst 方法，因为编译器无法确定参数类型。

通配符在类型系统中具有重要的意义，它们为一个泛型类所指定的类型集合提供了一个有用的类型范围。泛型参数表明的是在类、接口、方法的创建中，要使用一个数据类型参数来代表将来可能会用到的一种具体的数据类型。它可以是 Integer 类型，也可以是 String 类型。我们通常把它的类型定义成 E、T、K、V，等等。

当我们在实例化对象的时候，必须声明 T 具体是一个什么类型。所以当我们把

T定义成一个确定的泛型数据类型，参数就只能是这种数据类型。此时，我们就用到了通配符代替指定的泛型数据类型。

如果把一个对象分为声明、使用两部分的话。泛型主要是侧重于类型的声明的代码复用，通配符则侧重于使用上的代码复用。泛型用于定义内部数据类型的参数化，通配符则用于定义使用的对象类型的参数化。

使用泛型、通配符提高了代码的复用性。同时对象的类型得到了类型安全的检查，减少了类型转换过程中的错误。

6.2.2 泛型和数组的型变

在Java中数组是协变的，下面的代码是可以正确编译运行的：

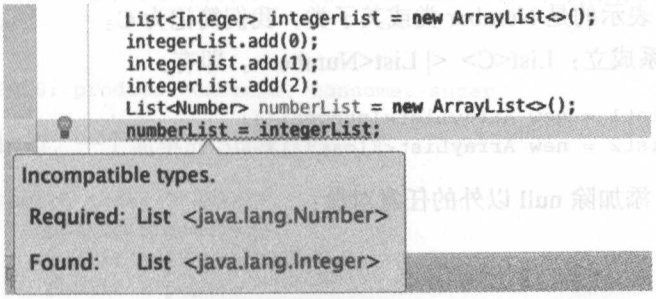
```
Integer[] ints = new Integer[3];
ints[0] = 0;
ints[1] = 1;
ints[2] = 2;
Number[] numbers = new Number[3];
numbers = ints;
for (Number n : numbers) {
    System.out.println(n);
}
```

在Java中，因为Integer是Number的子类型，数组类型Integer[]也是Number[]的子类型，因此在任何需要Number[]值的地方都可以提供一个Integer[]值。

而另一方面，泛型不是协变的。也就是说，List不是List的子类型，试图在要求List的位置提供List是一个类型错误。下面的代码，编译器是会直接报错的：

```
List<Integer> integerList = new ArrayList<>();
integerList.add(0);
integerList.add(1);
integerList.add(2);
List<Number> numberList = new ArrayList<>();
numberList = integerList;
```

编译器报错提示如下：

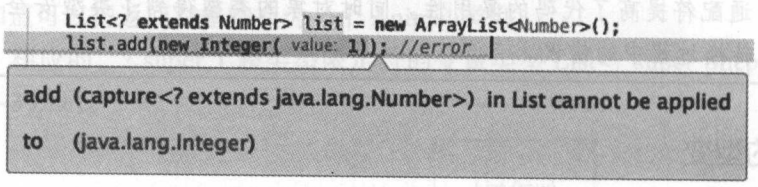


Java 中泛型和数组的不同行为的确引起了许多混乱。

就算我们使用通配符，这样写：

```
List<? extends Number> list = new ArrayList<Number>();
list.add(new Integer(1)); //error
```

仍然是报错的：



为什么 Number 的对象可以由 Integer 实例化，而 ArrayList 的对象却不能由 ArrayList 实例化？ list 中的 <? extends Number> 声明其元素是 Number 或 Number 的派生类，为什么不能 add Integer? 为了解决这些问题，需要了解 Java 中的逆变和协变以及泛型中通配符用法。

Animal 类型（简记为 F, Father）是 Dog 类型（简记为 C, Child）的父类型，我们把这种父子类型关系简记为 F <| C。

而 List, List 的类型，我们分别简记为 f(F), f(C)。

那么我们可以这么来描述协变和逆变：

当 F <| C 时，如果有 f(F) <| f(C)，那么 f 叫做协变；

当 F <| C 时，如果有 f(C) <| f(F)，那么 f 叫做逆变。

如果上面两种关系都不成立则叫做不变。

协变和逆协变都是类型安全的。

Java 中泛型是不变的，可有时需要实现逆变与协变，怎么办呢？这时就需要使用我们上面讲的通配符 ?。

1. <? extends T> 实现了泛型的协变

```
List<? extends Number> list = new ArrayList<>();
```

上例中的 ? extends Number 表示的是 Number 类或其子类，我们简记为 C。

这里 C <| Number，这个关系成立：List<C> <| List<Number>。即有：

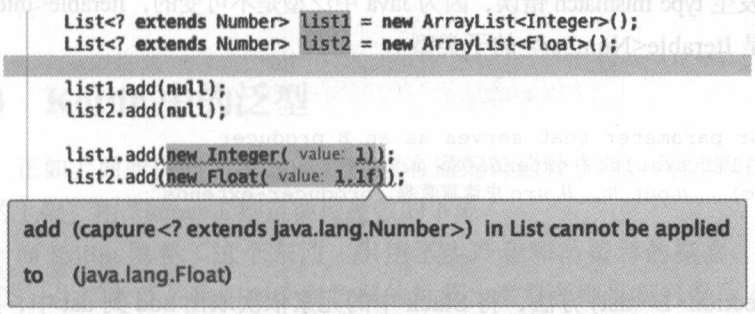
```
List<? extends Number> list1 = new ArrayList<Integer>();
List<? extends Number> list2 = new ArrayList<Float>();
```

但是这里不能向 list1、list2 添加除 null 以外的任意对象：

```
list1.add(null);
list2.add(null);
```

```
list1.add(new Integer(1));  //error
list2.add(new Float(1.1f)); //error
```

因为，List可以添加Integer及其子类，List可以添加Float及其子类，List、List都是List<? extends Number>的子类型，如果能将Float的子类添加到List<? extends Number>中，那么也能将Integer的子类添加到List<? extends Number>中，那么这时候List<? extends Number>里面将会持有各种Number子类型的对象（Byte、Integer、Float、Double等）。Java为了保护其类型一致，禁止向List<? extends Number>添加任意对象，不过可以添加null：



2. <? super T>实现了泛型的逆变

```
List<? super Number> list = new ArrayList<>();
```

上例中的？super Number通配符表示的类型下界为Number，即这里的父类型F是？super Number，子类型C是Number。即当F <| C，有f(C) <| f(F)，这就是逆变。代码示例：

```
List<? super Number> list3 = new ArrayList<Number>();
List<? super Number> list4 = new ArrayList<Object>();
list3.add(new Integer(3));
list4.add(new Integer(4));
```

也就是说，我们不能往List<? super Number>中添加Number的任意父类对象。但是可以向List<? super Number>添加Number及其子类对象。

3. PECS

现在问题来了：我们什么时候用extends，什么时候用super呢？《Effective Java》给出了答案：

```
PECS: producer-extends, consumer-super
```

比如，一个简单的Stack API：

```
public class Stack<E>{
    public Stack();
    public void push(E e):
    public E pop();
```

```
    public boolean isEmpty();
}
```

要实现 pushAll(Iterable src) 方法，将 src 的元素逐一入栈：

```
public void pushAll(Iterable<E> src){
    for(E e : src)
        push(e)
}
```

假设有一个实例化 Stack<Number> 的对象 stack，src 有 Iterable<Integer> 与 Iterable<Float>；在调用 pushAll 方法时会发生 type mismatch 错误，因为 Java 中泛型是不可变的，Iterable<Integer> 与 Iterable<Float> 都不是 Iterable<Number> 的子类型。

因此，应改为：

```
// Wildcard type for parameter that serves as an E producer
public void pushAll(Iterable<? extends E> src) {
    for (E e : src)    // out T, 从 src 中读取数据, producer-extends
        push(e);
}
```

要实现 popAll(Collection<E> dst) 方法，将 Stack 中的元素依次取出 add 到 dst 中，如果不用通配符实现：

```
// popAll method without wildcard type - deficient!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

同样，假设有一个实例化 Stack<Number> 的对象 stack，dst 为 Collection<Object>；调用 popAll 方法时会发生 type mismatch 错误，因为 Collection<Number> 不是 Collection<Object> 的子类型。

因而，应改为：

```
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E> dst) {
    while (!isEmpty())
        dst.add(pop());    // in T, 向 dst 中写入数据, consumer-super
}
```

Naftalin 与 Wadler 将 PECS 称为 Get and Put Principle。

在 java.util.Collections 的 copy 方法中 (JDK1.7) 完美地诠释了 PECS：

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    int srcSize = src.size();
    if (srcSize > dest.size())
        throw new IndexOutOfBoundsException("Source does not fit in dest");

    if (srcSize < COPY_THRESHOLD ||
```

```
        (src instanceof RandomAccess && dest instanceof RandomAccess)) {
        for (int i=0; i<srcSize; i++)
            dest.set(i, src.get(i));
    } else {
        ListIterator<? super T> di=dest.listIterator();   //in T, 写入dest数据
        ListIterator<? extends T> si=src.listIterator();  //out T, 读取src数据
        for (int i=0; i<srcSize; i++) {
            di.next();
            di.set(si.next());
        }
    }
}
```

6.3 Kotlin 中的泛型

正如上面所讲的，在 Java 泛型里有通配符这种东西，我们要用 ? extends T 指定类型参数的上限，用 ? super T 指定类型参数的下限。

而 Kotlin 抛弃了这个东西，引用了生产者和消费者的概念，也就是我们前面讲到的 PECS。生产者就是我们去读取数据的对象，消费者则是我们要写入数据的对象。这两个概念理解起来有点绕。

我们用代码示例简单讲解一下：

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    ...
    ListIterator<? super T> di=dest.listIterator();   //in T, 写入dest数据
    ListIterator<? extends T> si=src.listIterator();  //out T, 读取src数据
    ...
}
```

List<? super T> dest 是消费数据的对象，这些数据会写入到该对象中，这些数据该对象被“吃掉”了（Kotlin 中叫 in T）。

List<? extends T> src 是生产提供数据的对象。这些数据哪里来的呢？就是通过 src 读取获得的（Kotlin 中叫 out T）。

6.3.1 out T 与 in T

在 Kotlin 中，我们把那些只能保证读取数据时类型安全的对象叫做生产者，用 out T 标记；把那些只能保证写入数据安全时类型安全的对象叫做消费者，用 in T 标记。

提示 如果你觉得太晦涩难懂，就这么记吧：

out T 等价于 ? extends T

in T 等价于 ? super T

此外，还有 * 等价于 ?

6.3.2 声明处型变

Kotlin 泛型中添加了声明处型变。看下面的例子：

```
interface Source<out T> {
    fun <T> nextT();
}
```

我们在接口的声明处用 out T 做了生产者声明以实现安全的类型协变：

```
fun demo(str: Source<String>) {
    val obj: Source<Any> = str //合法的类型协变
}
```

Kotlin 中有大量的声明处协变，比如 Iterable 接口的声明：

```
public interface Iterable<out T> {
    public operator fun iterator(): Iterator<T>
}
```

因为 Collection 接口和 Map 接口都继承了 Iterable 接口，而 Iterable 接口被声明为生产者接口，所以所有的 Collection 和 Map 对象都可以实现安全的类型协变：

```
val c: List<Number> = listOf(1, 2, 3)
```

这里的 listOf() 函数返回 List<Int> 类型，因为 List 接口实现了安全的类型协变，所以可以安全地把 List<Int> 类型赋给 List<Number> 类型变量。

6.3.3 类型投影

将类型参数 T 声明为 out 非常方便，但是有些类实际上不能限制为只返回 T。

一个很好的例子是 Array：

```
class Array<T>(val size: Int) {
    fun get(index: Int): T { }
    fun set(index: Int, value: T) { }
}
```

该类在 T 上既不能是协变的也不能是逆变的，这造成了一些不灵活性。考虑下述函数：

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

这个函数应该将项目从一个数组复制到另一个数组。如果我们采用如下方式使用这个函数：

```
val ints: Array<Int> = arrayOf(1, 2, 3)
```

```
val any = Array<Any>(3) { "" }
copy(ints, any) // 错误：期望 (Array<Any>, Array<Any>)
```

这里我们将遇到同样的问题：Array <T> 在 T 上是不型变的，因此 Array <Int> 和 Array <Any> 都不是另一个的子类型。

那么，我们唯一要确保的是 copy() 不会做任何坏事。我们阻止它写到 from，我们可以：

```
fun copy(from: Array<out Any>, to: Array<Any>) {}
```

现在这个 from 是一个受 Array<out Any> 限制的（投影的）数组。在 Kotlin 中，称为类型投影（type projection）。其主要作用是参数作限定，避免不安全操作。

类似的，我们也可以使用 in 投影一个类型：

```
fun fill(dest: Array<in String>, value: String) {}
```

Array<in String> 对应于 Java 的 Array<? super String>，也就是说，我们可以传递一个 CharSequence 数组或一个 Object 数组给 fill() 函数。

提示 类似 Java 中的无界类型通配符 ?, Kotlin 也有对应的星投影语法 *。

例如，如果类型被声明为 interface Function <in T, out U>，我们有以下星投影：

- Function<*, String> 表示 Function<in Nothing, String>。
- Function<Int, *> 表示 Function<Int, out Any?>。
- Function<*, *> 表示 Function<in Nothing, out Any?>。

“* 投影”跟 Java 的原始类型类似，不过“* 投影”是安全的。

6.4 泛型类

声明一个泛型类：

```
class Box<T>(t: T) {
    var value = t
}
```

通常，要创建这样一个类的实例，我们需要指定类型参数：

```
val box: Box<Int> = Box<Int>(1)
```

但是，如果类型参数可以通过推断得到，比如，通过构造器参数类型，或通过其他手段推断得到，此时允许省略类型参数：

```
val box = Box(1) // 1 的类型为 Int，因此编译器知道我们创建的实例是 Box<Int> 类型
```

6.5 泛型函数

类可以有类型参数，函数也有。类型参数要放在函数名称之前：

```
fun <T> singletonList(item: T): List<T> {}
fun <T> T.basicToString() : String {  //扩展函数
}
```

要调用泛型函数，在函数名后指定类型参数即可：

```
val l = singletonList<Int>(1)
```



泛型函数与其所在的类是否是泛型没有关系。泛型函数独立于其所在的类。我们应该尽量使用泛型方法，也就是说，如果使用泛型方法可以取代将整个类泛型化，那么就应该只使用泛型方法，因为它可以使事情更明白。

6.6 本章小结

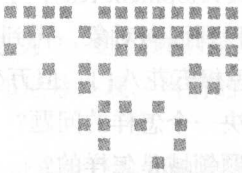
泛型是一个非常有用的东西。在面向对象的集合中处处体现泛型和多态的思想。在Kotlin的集合类中，我们可以发现泛型代码几乎无处不在。这些泛型代码可以使得我们写的函数、类更加通用。

Kotlin的泛型思想跟Java基本相似。不过，Kotlin把Java中的型变、泛型通配符等用法提炼优化了（in/out），使得我们写泛型代码更加方便了。

泛型不范。什么意思呢？就是说，泛型变量其实是对Java/Kotlin类型及其行为的一种“抽象”，或者说是“限定”，这使得我们的代码类型更加清晰有序且安全。

Kotlin是一门支持面向对象编程（OOP）跟函数式编程（FP）的强大语言。我们已经学习了Kotlin的语言基础知识、类型系统、集合类、泛型等相关知识了，相信你已经对Kotlin有了一个初步的了解。后面，我们将一起来学习Kotlin的面向对象编程和函数式编程。

本章示例代码：https://github.com/EasyKotlin/chapter6_generics



第 7 章 Chapter 7

面向对象编程

在前面的章节中，我们学习了 Kotlin 的语言基础知识、类型系统、集合类以及泛型相关的知识。在本章节以及下一章中，我们将一起来学习 Kotlin 对面向对象编程以及函数式编程的支持。

本章我们主要介绍 Kotlin 面向对象编程的特性：类与构造函数、抽象类与接口、继承以及多重继承等基础知识，同时介绍了 Kotlin 中的注解类、枚举类、数据类、密封类、嵌套类、内部类、匿名内部类等特性类。最后我们学习了 Kotlin 中对单例模式、委托模式的语言层面上的内置支持：object 对象、委托。

下面就让我们从面向对象编程的基本思想开始。

7.1 面向对象编程思想

7.1.1 一切皆是映射

《易传 · 系辞上传》写道：“易有太极，是生两仪，两仪生四象，四象生八卦。”如今的互联网世界，其基石却是 01（阴阳），不得不佩服我华夏先祖的智慧博大精深。

计算机领域中的所有问题，都可以通过向上一层进行抽象封装来解决。这里提及的封装的本质，其实就是“映射”。

就好比通过的电子电路中的电平进行 01 逻辑映射，于是有了布尔代数、数字逻辑电路系统；对 01 逻辑的进一步封装抽象成 CPU 指令集映射，诞生了汇编语言；通过汇编语言的向上抽象一层编译解释器，于是有了 Pascal、Fortran、C 语言；再对核心函数 API 进行封装

形成开发包（Development Kit），于是有了 Java、C++ 语言。

从面向过程到面向对象，再到设计模式、架构设计、面向服务、Sass/Pass/Iass 等思想，各种软件理论思想五花八门，但万变不离其宗，离不开如下问题：

- 你要解决一个怎样的问题？
- 你的问题领域是怎样的？
- 你的模型（数据结构）是什么？
- 你的算法是什么？
- 你对这个世界的本质认知是怎样的？
- 你的业务领域的逻辑问题，流程是什么？

等等。

著名程序大师 Grady Booch 说：我对 OO 编程的目标从来就不是复用。相反，对我来说，对象提供了一种处理复杂性的方式。这个问题可以追溯到亚里士多德：你把这个世界视为过程还是对象？在 OO 兴起运动之前，编程以过程为中心。例如结构化设计方法。然而，系统已经到达了超越其处理能力的复杂性极点。有了对象，我们能够通过提升抽象级别来构建更大的、更复杂的系统，我认为，这才是面向对象编程运动的真正胜利。

最初，人们使用物理的或逻辑的二进制机器指令来编写程序，尝试着表达思想中的逻辑，控制硬件计算和显示，发现是可行的；接着，创造了助记符——汇编语言，比机器指令更容易记忆；再接着，创造了编译器、解释器和计算机高级语言，能够以人类友好自然的方式去编写程序，在牺牲少量性能的情况下，获得比汇编语言更强且更容易使用的语句控制能力：条件、分支、循环，以及更多的语言特性：指针、结构体、联合体、枚举等，还创造了函数，能够将一系列指令封装成一个独立的逻辑块反复使用；逐渐地，产生了面向过程的编程方法；后来，人们发现将数据和逻辑封装成对象，更接近于现实世界，且更容易维护大型软件，又出现了面向对象的编程语言和编程方法学，增加了新的语言特性：继承、多态、模板、异常错误。

为了不必重复开发常见工具和任务，人们创造和封装了容器及算法、SDK，垃圾回收器，甚至是并发库；为了让计算机语言更有力更有效率地表达各种现实逻辑，消解软件开发中遇到的冲突，还在语言中支持了元编程、高阶函数、闭包等有用特性。

为了更高效率地开发可靠的软件和应用程序，人们逐渐构建了代码编辑器、IDE、代码版本管理工具、公共库、应用框架、可复用组件、系统规范、网络协议、语言标准等，针对遇到的问题提出了许多不同的思路和解决方案，并总结提炼成特定的技术和设计模式，还探讨和形成了不少软件开发过程，用来保证最终发布的软件质量。尽管编写的这些软件和工具还存在不少 BUG，但是它们都“奇迹般地存活”，并共同构建了今天蔚为壮观的互联网时代的电商，互联网金融、云计算、大数据、物联网、机器智能等等的“虚拟世界”。

7.1.2　二进制 01 与易经阴阳

二进制数是用 0 和 1 两个数码来表示的数。它的基数为 2，进位规则是“逢二进一”，借位规则是“借一当二”，由 18 世纪德国数理哲学大师莱布尼兹发现。当前的计算机系统使用的基本上是二进制系统。

19 世纪爱尔兰逻辑学家乔治·布尔对逻辑命题的思考过程转化为对符号 0、1 的某种代数演算，二进制是逢 2 进位的进位制。0、1 是基本算符。因为它只使用 0、1 两个数字符号，非常简单方便，易于用电子方式实现。

二进制的发现直接导致了电子计算器和计算机的发明，并让计算机得到了迅速的普及，进入各行各业，成为人类生活和生产的重要工具。

二进制的实质是通过两个数字“0”和“1”来描述事件。在人类的生产、生活等许多领域，我们可以通过计算机来虚拟地描述现实中存在的事件，并能通过给定的条件和参数模拟事件变化的规律。二进制的计算机几乎是万能的，能将我们生活的现实世界完美复制，并且还能根据人类给定的条件模拟在现实世界难以实现的各种实验。

但是，不论计算机能给我们如何多变、如何完美、如何复杂的画面，其本源只是简单的“0”和“1”。“0”和“1”在计算机中通过不同的组合与再组合，模拟出一个纷繁复杂、包罗万象的虚拟世界。如图 7-1 所示。

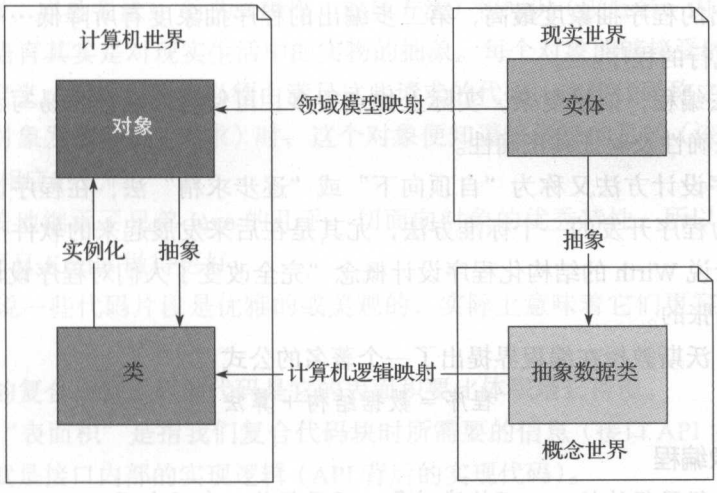


图 7-1　计算机世界

二进制的“0”和“1”通过计算机里能够创造出一个虚拟的、纷繁的世界。自然界中的阴阳形成了现实世界的万事万物。

所以自然世界的“阴”“阳”作为基础切实地造就了复杂的现实世界，计算机的“0”和“1”形象地模拟现实世界的一切现象，易学中的“卦”和“阴阳爻”抽象地揭示了自然界存在的事件和其变化规律。

所以说，编程的本质跟大自然创造万物的本质是一样的。

7.1.3 从面向过程到面向对象

从IBM公司的约翰·巴库斯在1957年开发出世界上第一个高级程序设计语言Fortran至今，高级程序设计语言的发展已经经历了整整半个世纪。在这期间，程序设计语言主要经历了从面向过程（如C和Pascal语言）到面向对象（如C++和Java语言），再到面向组件编程（如.NET平台下的C#语言），以及面向服务架构技术（如SOA、Service以及最近很火的微服务架构）等。

1. 面向过程编程

结构化编程思想的核心：功能分解（自顶向下，逐步求精）。

1971年4月份的Communications of ACM上，尼古拉斯·沃斯（Niklaus Wirth，1934年2月15日—，结构化编程思想的创始人。因发明了Euler、Alogo-W、Modula和Pascal等一系列优秀的编程语言并提出了结构化编程思想而在1984年获得了图灵奖。）发表了论文“通过逐步求精方式开发程序”（Program Development by Stepwise Refinement），首次提出“结构化程序设计”（structure programming）的概念。

不要求一步就编制成可执行的程序，而是分若干步进行，逐步求精。

第一步编出的程序抽象度最高，第二步编出的程序抽象度有所降低……最后一步编出的程序即为可执行的程序。

用这种方法编程，似乎复杂，实际上优点很多，可使程序易读、易写、易调试、易维护、易保证其正确性及验证其正确性。

结构化程序设计方法又称为“自顶向下”或“逐步求精”法，在程序设计领域引发了一场革命，成为程序开发的一个标准方法，尤其是在后来发展起来的软件工程中获得广泛应用。有人评价说Wirth的结构化程序设计概念“完全改变了人们对程序设计的思维方式”，这是一点也不夸张的。

尼古拉斯·沃斯教授在编程界提出了一个著名的公式：

程序 = 数据结构 + 算法

2. 面向对象编程

面向对象编程思想的核心：万物皆对象。顶层架构，自顶向下。

阿伦·凯（Alan Kay）：面向对象编程思想的创始人。2003年因在面向对象编程上所做的巨大贡献而获得图灵奖。他的名言是：

The best way to predict the future is to invent it，预测未来最好的方法是创造它！

阿伦·凯是Smalltalk面向对象编程语言的发明人之一，也是面向对象编程思想的创始

人之一，同时，他还是笔记本电脑最早的构想者和现代 Windows GUI 的建筑师。最早提出 PC 概念和互联网的也是阿伦·凯，所以人们都尊称他为“预言大师”。他是当今 IT 界屈指可数的技术天才级人物。

面向对象编程思想主要是复用性和灵活性（弹性）。复用性是面向对象编程的一个主要机制。灵活性主要是应对变化的特性，因为客户的需求是不断改变的，怎样适应客户需求的变化，这是软件设计灵活性或者说是弹性的问题。

Java 是一种面向对象编程语言，它基于 Smalltalk 语言，作为 OOP 语言，它具有以下五个基本特性：

1）万物皆对象，每一个对象都会存储数据，并且可以对自身执行操作。因此，每一个对象包含两部分：成员变量和成员方法。在成员方法中可以改变成员变量的值。

2）程序是对象的集合，他们通过发送消息来告知彼此所要做的事情，也就是调用相应的成员函数。

3）每一个对象都有自己的由其他对象所构成的存储，也就是说在创建新对象的时候可以在成员变量中使用已存在的对象。

4）每个对象都拥有其类型，每个对象都是某个类的一个实例，每一个类区别于其他类的特性就是可以向它发送什么类型的消息，也就是它定义了哪些成员函数。

5）某一个特定类型的所有对象都可以接受同样的消息。另一种对对象的描述为：对象具有状态（数据，成员变量）、行为（操作，成员方法）和标识（成员名，内存地址）。

面向对象语言其实是对现实生活中的实物的抽象。每个对象能够接受的请求（消息）由对象的接口所定义，而在程序中必须由满足这些请求的代码，这段代码称之为这个接口的实现。当向某个对象发送消息（请求）时，这个对象便知道该消息的目的（该方法的实现已定义），然后执行相应的代码。

Kotlin 完美地继承了兄弟 Java 的几乎一切面向对象的优秀特性，所以上面的这些特性 Kotlin 都有，而且 Kotlin 做得更好。

我们经常说一些代码片段是优雅的或美观的，实际上意味着它们更容易被人类有限的思维所处理。

对于程序的复合而言，好的代码是它的表面积要比体积增长得慢。

代码块的“表面积”是指我们复合代码块时所需要的信息（接口 API 协议定义）。代码块的“体积”就是接口内部的实现逻辑（API 背后的实现代码）。

在面向对象编程中，一个理想的对象应该是只暴露它的抽象接口（纯表面，无体积），其方法则扮演箭头的角色。如果为了理解一个对象如何与其他对象进行复合，当你发现不得不深入挖掘对象的实现时，你所用的编程范式的原本优势就荡然无存了。

3. 面向组件和面向服务

面向组件——我们知道面向对象支持重用，但是重用的单元很小，一般是类；而面向组件则不同，它可以重用多个类甚至一个程序。也就是说面向组件支持更大范围内的重用，开

发效率更高。如果把面向对象比作重用零件，那么面向组件则是重用部件。

面向服务——将系统进行功能化，每个功能提供一种服务。现在非常流行微服务（MicroService）技术以及SOA（面向服务架构）技术都是面向服务的。

最后，再往大了说，整个的软件、平台、基础设施，甚至全世界的互联网系统，都是一种服务。

抽象层次从低到高，我们可以大致概括如下：

面向过程（Procedure）→面向对象（Object）→面向组件（Component）→面向服务（Service）→SaaS/PasS/IasS →互联网系统

正如解决数学问题通常我们会谈“思想”，诸如反证法、化繁为简等，解决计算机问题也有很多非常出色的思想。思想之所以称为思想，是因为“思想”有拓展性与引导性，可以解决一系列问题。

解决问题的复杂程度直接取决于抽象的种类及质量。如果将结构、性质不同的底层实现进行封装，向上提供统一的API接口，让使用者觉得就是在使用一个统一的资源，或者让使用者觉得自己在使用一个本来底层不直接提供、“虚拟”出来的资源。

计算机中的所有问题，都可以通过向上抽象封装一层来解决。同样，任何复杂的问题，最终总能够回归最本质，最简单的因素。

面向对象编程是一种自顶向下的程序设计方法。万事万物都是对象，对象有其行为（方法），状态（成员变量，属性）。OOP是一种编程思想，而不是针对某个语言的。当然，语言影响思维方式，思维依赖语言的表达，这也是辩证的来看。

所谓“面向对象语言”，其实经典的“过程式语言”（比如Pascal、C），也能体现面向对象的思想。所谓“类”和“对象”，就是C语言里面的抽象数据类型结构体（struct）。

而多态是唯一面向对象相比struct多付出的代价，也是最重要的特性。这就是SmallTalk、Java这样的面向对象语言所提供的特性。



回到一个古老的话题：程序是什么？

在面向对象的编程世界里，下面的这个公式：

程序 = 算法 + 数据结构

可以简单重构成：

程序 = 基于对象操作的算法 + 以对象为最小单位的数据结构

封装总是为了减少操作粒度，数据结构上的封装导致了数据的减少，自然减少了问题求解的复杂度；对代码的封装使得代码得以复用，减少了代码的体积，同样使问题简化。这个时候，算法操作的就是一个抽象概念的集合。

在面向对象的程序设计中，我们便少不了集合类容器。容器就用来存放一类有共同抽

象概念的东西。这里说有共同概念的东西（而没有说对象），其实，就是我们上一个章节中讲到的泛型。这样对于一个通用的算法，我们就可以最大化地实现复用，作用于集合。

面向对象的本质就是让对象有多态性，把不同对象以同一特性来归组，统一处理。至于所谓继承父类、实现接口等概念，只是多态性的实现细节有不同。

在遵循这些面向对象设计原则基础上，前辈们总结出一些解决不同问题场景的设计模式，以 GoF 的 23 种设计模式最为知名。

我们用一幅图简单概括一下面向对象编程的知识框架，如图 7-2 所示。

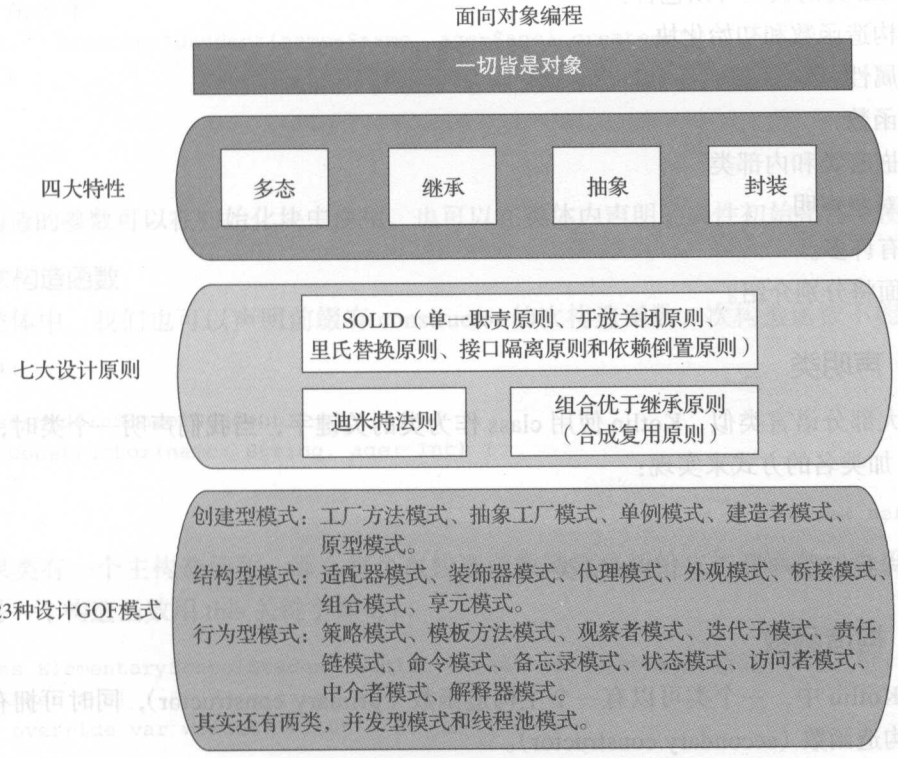


图 7-2　面向对象编程的知识框架

讲了这么多思想层面的东西，我们下面来开始 Kotlin 的面向对象编程的学习。Kotlin 对面向对象编程是完全支持的。

7.2　类与构造函数

下面我们来一起学习 Kotlin 的面向对象的特性。

Kotlin 和 Java 很相似，也是一种面向对象的语言。如果你熟悉 Java 或者 C++、C# 中的类，你可以很快上手。同时，你也将看到 Kotlin 与 Java 中的面向对象编程的一些不同的特性。

Kotlin 中的类和接口跟 Java 中对应的概念有些不同，比如接口可以包含属性声明；Kotlin 的类声明，默认是 final 和 public 的。

另外，嵌套类并不是默认在内部的。它们不包含外部类的隐式引用。

在构造函数方面，Kotlin 简短的主构造函数在大多数情况下都可以满足使用，当然如果有稍微复杂的初始化逻辑，我们也可以声明次级构造函数来完成。

我们还可以使用 data 修饰符来声明一个数据类，使用 object 关键字来表示单例对象、伴生对象等：

Kotlin 类的成员可以包含：

- ❑ 构造函数和初始化块
- ❑ 属性
- ❑ 函数
- ❑ 嵌套类和内部类
- ❑ 对象声明

还有许多。

下面将分别介绍。

7.2.1 声明类

和大部分语言类似，Kotlin 使用 class 作为类的关键字，当我们声明一个类时，直接通过 class 加类名的方式来实现：

```
class World
```

这样我们就声明了一个 World 类。

7.2.2 构造函数

在 Kotlin 中，一个类可以有一个主构造函数（primary constructor），同时可拥有一个或多个次构造函数（secondary constructor）。

1. 主构造函数

主构造函数是类头的一部分，直接放在类名后面：

```
open class Student constructor(var name: String, var age: Int) : Any() {
...
}
```

如果主构造函数没有任何注解或者可见性修饰符，可以省略 constructor 关键字。如果构造函数有注解或可见性修饰符，constructor 关键字是必需的，并且这些修饰符在它前面：

```
annotation class MyAutowired

class ElementaryStudent public @MyAutowired constructor(name: String, age: Int) :
```

```
    Student(name, age) {
...
}
```

与普通属性一样，主构造函数中声明的属性可以是可变的（var）或只读的（val）。

主构造函数不能包含任何的代码。初始化的代码可以放到以 init 关键字作为前缀的初始化块（initializer blocks）中：

```
open class Student constructor(var name: String, var age: Int) : Any() {

    init {
        println("Student{name=$name, age=$age} created!")
    }
    ...

}
```

主构造的参数可以在初始化块中使用，也可以在类体内声明的属性初始化器中使用。

2. 次构造函数

在类体中，我们也可以声明前缀有 constructor 的次构造函数，次构造函数不能有声明 val 或 var：

```
class MiddleSchoolStudent {
    constructor(name: String, age: Int) {
    }
}
```

如果类有一个主构造函数，那么每个次构造函数需要委托给主构造函数，委托到同一个类的另一个构造函数用 this 关键字即可：

```
class ElementarySchoolStudent public @MyAutowired constructor(name: String,
    age: Int) : Student(name, age) {
    override var weight: Float = 80.0f

    constructor(name: String, age: Int, weight: Float) : this(name, age) {
        this.weight = weight
    }

    ...
}
```

如果一个非抽象类没有声明任何（主或次）构造函数，它会有一个生成的不带参数的主构造函数。构造函数的可见性是 public。

3. 私有主构造函数

我们如果希望这个构造函数是私有的，我们可以如下声明：

```
class DontCreateMe private constructor() {
}
```

这样我们在代码中，就无法直接使用主构造函数来实例化这个类，下面的写法是不允许的：

```
val dontCreateMe = DontCreateMe() //cannot access it
```

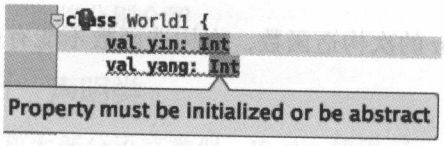
但是，我们可以通过次构造函数引用这个私有主构造函数来实例化对象：

7.2.3 类的属性

我们再给这个 World 类加入两个属性。我们可能直接简单地写成：

```
class World1 {
    val yin: Int
    val yang: Int
}
```

在 Kotlin 中，直接这样写语法上是会报错的：



意思很明显，是说这个类的属性必须要初始化，或者如果不初始化那就得是抽象的 abstract 属性。

我们把这两个属性都给初始化如下：

```
class World1 {
    val yin: Int = 0
    val yang: Int = 1
}
```

再使用测试代码来看一下访问这两个属性的方式：

```
>>> class World1 {
...     val yin: Int = 0
...     val yang: Int = 1
... }
>>> val w1 = World1()
>>> w1.yin
0
>>> w1.yang
1
```

上面的 World1 类的代码，在 Java 中等价的写法是：

```
public final class World1 {
```

```
    private final int yin;
    private final int yang = 1;

    public final int getYin() {
        return this.yin;
    }

    public final int getYang() {
        return this.yang;
    }
}
```

我们可以看出，Kotlin 中的类的字段自动带有 getter 方法和 setter 方法。而且写起来比 Java 要简洁得多。

7.2.4　函数（方法）

我们再来给这个 World1 类中加上一个函数：

```
class World2 {
    val yin: Int = 0
    val yang: Int = 1

    fun plus(): Int {
        return yin + yang
    }
}


val w2 = World2()
println(w2.plus()) //输出 1
```

7.3　抽象类

7.3.1　抽象类的定义

含有抽象函数的类（这样的类需要使用 abstract 修饰符来声明），称为抽象类。

下面是一个抽象类的例子：

```
abstract class Person(var name: String, var age: Int) : Any() {

    abstract var addr: String
    abstract val weight: Float

    abstract fun doEat()
    abstract fun doWalk()

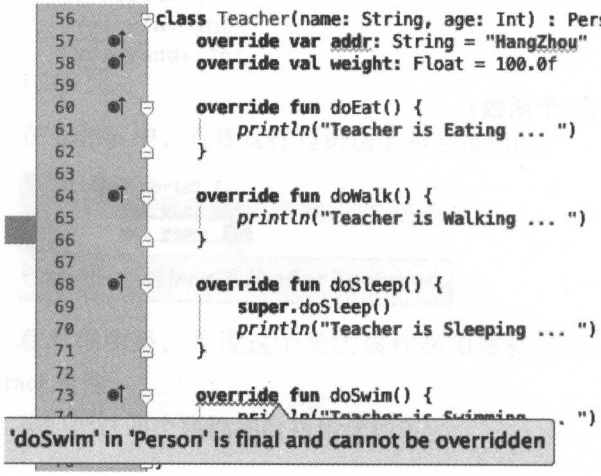
    fun doSwim() {
```

```
        println("I am Swimming ... ")
    }

    open fun doSleep() {
        println("I am Sleeping ... ")
    }
}
```

7.3.2 抽象函数

在上面的这个抽象类中，不仅可以有抽象函数 abstract fun doEat() abstract fun doWalk()，同时可以有具体实现的函数 fun doSwim()，这个函数默认是 final 的。也就是说，我们不能重写这个 doSwim 函数：



如果一个函数想要设计成能被重写，例如 fun doSleep()，我们给它加上 open 关键字即可。然后，就可以在子类中重写这个 open fun doSleep()：

```
class Teacher(name: String, age: Int) : Person(name, age) {
    override var addr: String = "HangZhou"
    override val weight: Float = 100.0f

    override fun doEat() {
        println("Teacher is Eating ... ")
    }

    override fun doWalk() {
        println("Teacher is Walking ... ")
    }

    override fun doSleep() {
        super.doSleep()
        println("Teacher is Sleeping ... ")
    }
```

```
//    override fun doSwim() { //cannot be overriden
//        println("Teacher is Swimming ... ")
//    }
}
```

抽象函数是一种特殊的函数：它只有声明，而没有具体的实现。抽象函数的声明格式为：

```
abstract fun doEat()
```



关于抽象函数的特征，我们简单总结如下：

- 抽象函数必须用 abstract 关键字进行修饰。
- 抽象函数不用手动添加 open，默认被 open 修饰。
- 抽象函数没有具体的实现。
- 含有抽象函数的类成为抽象类，必须由 abtract 关键字修饰。抽象类中可以有具体实现的函数，这样的函数默认是 final(不能被覆盖重写)，如果想要重写这个函数，给这个函数加上 open 关键字。

7.3.3　抽象属性和抽象类

抽象属性就是在 var 或 val 前被 abstract 修饰，抽象属性的声明格式为：

```
abstract var addr : String
abstract val weight : Float
```



关于抽象属性，需要注意以下两点：

1）抽象属相在抽象类中不能被初始化。

2）如果在子类中没有主构造函数，要对抽象属性手动初始化。如果子类中有主构造函数，抽象属性可以在主构造函数中声明。

抽象类和普通类的区别有：

1）抽象函数必须为 public 或者 protected（因为如果为 private，则不能被子类继承，子类便无法实现该方法），默认情况为 public。

也就是说，这三个函数：

```
abstract fun doEat()
abstract fun doWalk()

fun doSwim() {
    println("I am Swimming ... ")
}
```

默认都是 public 的。

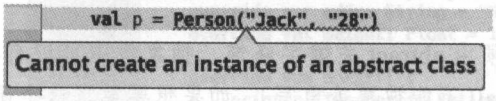
另外抽象类中的具体实现的函数，默认是 final 的。上面的三个函数，等价的 Java 的代码如下：

```
public abstract void doEat();

public abstract void doWalk();

public final void doSwim() {
    String var1 = "I am Swimming ... ";
    System.out.println(var1);
}
```

2）抽象类不能用来创建对象实例。也就是说，下面的写法编译器是不允许的：



3）如果一个类继承于一个抽象类，则子类必须实现父类的抽象方法。实现父类抽象函数，我们使用 override 关键字来表明是重写函数：

```
class Programmer(override var addr: String, override val weight: Float, name:
    String, age: Int) : Person(name, age) {
    override fun doEat() {
        println("Programmer is Eating ... ")
    }

    override fun doWalk() {
        println("Programmer is Walking ... ")
    }
}
```

如果子类没有实现父类的抽象函数，则必须将子类也定义为为 abstract 类。例如：

```
abstract class Writer(override var addr: String, override val weight: Float,
    name: String, age: Int) : Person(name, age) {
    override fun doEat() {
        println("Programmer is Eating ... ")
    }

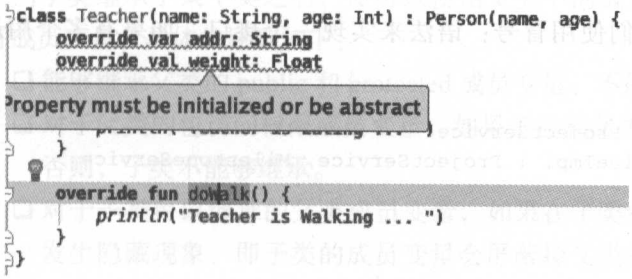
    abstract override fun doWalk();
}
```

doWalk 函数没有实现父类的抽象函数，那么我们在子类中把它依然定义为抽象函数。相应地这个子类，也成为了抽象子类，需要使用 abstract 关键字来声明。

如果抽象类中含有抽象属性，再实现子类中必须将抽象属性初始化，除非子类也为抽象类。例如我们声明一个 Teacher 类继承 Person 类：

```
class Teacher(name: String, age: Int) : Person(name, age) {
    override var addr: String  // error, 需要初始化, 或者声明为 abstract
    override val weight: Float // error, 需要初始化, 或者声明为 abstract
    ...
}
```

这样写，编译器会直接报错：



解决方法是，在实现的子类中，我们将抽象属性初始化即可：

```
class Teacher(name: String, age: Int) : Person(name, age) {
    override var addr: String = "HangZhou"
    override val weight: Float = 100.0f

    override fun doEat() {
        println("Teacher is Eating ... ")
    }

    override fun doWalk() {
        println("Teacher is Walking ... ")
    }
}
```

7.4　接口

和 Java 类似，Kotlin 使用 interface 作为接口的关键词。

7.4.1　接口定义

语法：

```
interface ProjectService
```

Kotlin 的接口与 Java 8 的接口类似。与抽象类相比，它们都可以包含抽象的方法以及方法的实现：

```
interface ProjectService {
    val name: String
    val owner: String
```

```
    fun save(project: Project)
    fun print() {
        println("I am project")
    }
}
```

7.4.2 实现接口

接口是没有构造函数的。我们使用冒号：语法来实现一个接口，如果有多个用逗号隔开：

```
class ProjectServiceImpl : ProjectService
class ProjectMilestoneServiceImpl : ProjectService, MilestoneService
```

我们也可以实现多个接口：

```
class Project

class Milestone

interface ProjectService {
    val name: String
    val owner: String
    fun save(project: Project)
    fun print() {
        println("I am project")
    }
}

interface MilestoneService {
    val name: String
    fun save(milestone: Milestone)
    fun print() {
        println("I am Milestone")
    }
}

class ProjectMilestoneServiceImpl : ProjectService, MilestoneService {
    override val name: String
        get() = "ProjectMilestone"
    override val owner: String
        get() = "Jack"

    override fun save(project: Project) {
        println("Save Project")
    }

    override fun print() {
//          super.print()
        super<ProjectService>.print()
        super<MilestoneService>.print()
```

```
    }

    override fun save(milestone: Milestone) {
        println("Save Milestone")
    }
}
```

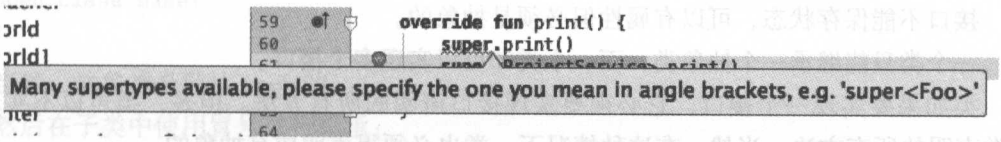
当子类继承了某个类之后，便可以使用父类中的成员变量，但是并不完全继承父类的所有成员变量。具体的原则如下：

- 能够继承父类的 public 和 protected 成员变量；不能够继承父类的 private 成员变量。
- 对于父类的包访问权限成员变量，如果子类和父类在同一个包下，则子类能够继承；否则，子类不能够继承。
- 对于子类可以继承的父类成员变量，如果在子类中出现了同名称的成员变量，则会发生隐藏现象，即子类的成员变量会屏蔽掉父类的同名成员变量。如果要在子类中访问父类中同名成员变量，需要使用 super 关键字来进行引用。

7.4.3　覆盖冲突

在 kotlin 中，实现继承通常遵循如下规则：如果一个类从它的直接父类继承了同一个函数的多个实现，那么它必须重写这个函数并且提供自己的实现。为表示使用父类中提供的方法，我们用 super 关键字表示。

在重写 print() 时，因为我们实现的 ProjectService、MilestoneService 都有一个 print() 函数，当我们直接使用 super.print() 时，编译器是无法知道想要调用的是哪个里面的 print 函数的，这个我们叫做覆盖冲突：



这个时候，我们可以使用下面的语法来调用：

```
super<ProjectService>.print()
super<MilestoneService>.print()
```

7.4.4　接口中的属性

在接口中声明的属性可以是抽象的，或者是提供访问器的实现。

在企业应用中，大多数的类型都是无状态的，如：Controller、ApplicationService、DomainService、Repository 等。

因为接口没有状态，所以它的属性是无状态的：

```
interface MilestoneService {
    val name: String //抽象的
```

```
    val owner: String get() = "Jack" //访问器

    fun save(milestone: Milestone)
    fun print() {
        println("I am Milestone")
    }
}

class MilestoneServiceImpl : MilestoneService {
    override val name: String
        get() = "MilestoneServiceImpl name"


    override fun save(milestone: Milestone) {
        println("save Milestone")
    }
}
```

7.5 抽象类和接口的差异

1. 概念上的区别

接口主要是对动作的抽象，定义了行为特性的规约。

抽象类是对根源的抽象。当你关注一个事物的本质的时候，用抽象类；当你关注一个操作的时候，用接口。

2. 语法层面上的区别

接口不能保存状态，可以有属性但必须是抽象的。

一个类只能继承一个抽象类，而一个类却可以实现多个接口。

类如果要实现一个接口，它必须要实现接口声明的所有方法。但是，类可以不实现抽象类声明的所有方法，当然，在这种情况下，类也必须得声明成是抽象的。

接口中所有的方法隐含的都是抽象的。而抽象类则可以同时包含抽象和非抽象的方法。

3. 设计层面上的区别

抽象类是对一种事物的抽象，即对类抽象，而接口是对行为的抽象。抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。

继承是“is a”的关系，而接口实现则是“has a”的关系。如果一个类继承了某个抽象类，则子类必定是抽象类的种类，而接口实现就不需要有这层类型关系。

设计层面不同，抽象类作为很多子类的父类，它是一种模板式设计。而接口是一种行为规范，它是一种辐射式设计。也就是说：

- 对于抽象类，如果需要添加新的方法，可以直接在抽象类中添加具体的实现，子类可以不进行变更。

- 而接口则不行，如果接口进行了变更，则所有实现这个接口的类都必须进行相应的改动。

4. 实际应用上的差异

在实际使用中，使用抽象类（也就是继承）是一种强耦合的设计，用来描述“A is a B”的关系，即如果说 A 继承于 B，那么在代码中将 A 当做 B 去使用应该完全没有问题。比如在 Android 中，各种控件都可以被当做 View 去处理。

如果在你设计中有两个类型的关系并不是“is a”，而是“is like a”，那就必须慎重考虑继承。因为一旦我们使用了继承，就要小心处理好子类跟父类的耦合依赖关系。此时，接口优于类继承。

7.6　继承

继承是面向对象编程的一个重要方式，因为通过继承，子类就可以扩展父类的功能。

在 Kotlin 中，所有的类会默认继承 Any 这个父类，但 Any 并不完全等同于 Java 中的 Object 类，因为它只有 equals()、hashCode() 和 toString() 这三个方法。

7.6.1　open 类

除了抽象类、接口默认可以被继承（实现）外，我们也可以把一个类声明为 open，这样我们就可以继承这个 open 类。

当我们想定义一个父类时，需要使用 open 关键字：

```
open class Base{
}
```

当然，抽象类是默认 open 的。

然后在子类中使用冒号进行继承：

```
class SubClass : Base(){
}
```

如果父类有构造函数，那么必须在子类的主构造函数中进行继承，没有的话则可以选择主构造函数或二级构造函数：

```
//父类
open class Base(type:String){

}

//子类
class SubClass(type:String) : Base(type){

}
```

Kotlin 中的 override（重写）和 Java 中也有所不同，因为 Kotlin 提倡所有的操作都是明确的，因此需要将希望被重写的函数设为 open：

```
open fun doSomething() {}
```

然后通过 override 标记实现重写：

```
override fun doSomething() {
    super.doSomething()
    }
```

同样，抽象函数以及接口中定义的函数默认都是 open 的。

重写的函数也是 open 的，如果希望它不被重写，可以在前面增加 final：

```
open class SubClass : Base{
    constructor(type:String) : super(type){
    }

    final override fun doSomething() {
    super.doSomething()
    }
}
```

7.6.2 多重继承

有些编程语言支持一个类拥有多个父类，例如 C++，我们将这个特性称为多重继承（multiple inheritance）。多重继承会有二义性和钻石型继承树（DOD：Diamond Of Death）的复杂性问题。Kotlin 跟 Java 一样，没有采用多继承，任何一个子类仅允许一个父类存在，而在多重继承的问题场景下，使用实现多个 interface 组合的方式来实现多重继承的功能。

代码示例：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/7/2.
 */
abstract class Animal {
    fun doEat() {
        println("Animal Eating")
    }
}


abstract class Plant {
    fun doEat() {
        println("Plant Eating")
    }
```

```
}

interface Runnable {
    fun doRun()
}

interface Flyable {
    fun doFly()
}

class Dog : Animal(), Runnable {
    override fun doRun() {
        println("Dog Running")
    }
}

class Eagle : Animal(), Flyable {
    override fun doFly() {
        println("Eagle Flying")
    }
}

//始祖鸟，能飞也能跑
class Archaeopteryx : Animal(), Runnable, Flyable {
    override fun doRun() {
        println("Archaeopteryx Running")
    }

    override fun doFly() {
        println("Archaeopteryx Flying")
    }

}

fun main(args: Array<String>) {
    val d = Dog()
    d.doEat()
    d.doRun()

    val e = Eagle()
    e.doEat()
    e.doFly()

    val a = Archaeopteryx()
    a.doEat()
    a.doFly()
    a.doRun()
}
```

上述代码类之间的关系，我们用图示表达，见图 7-3。

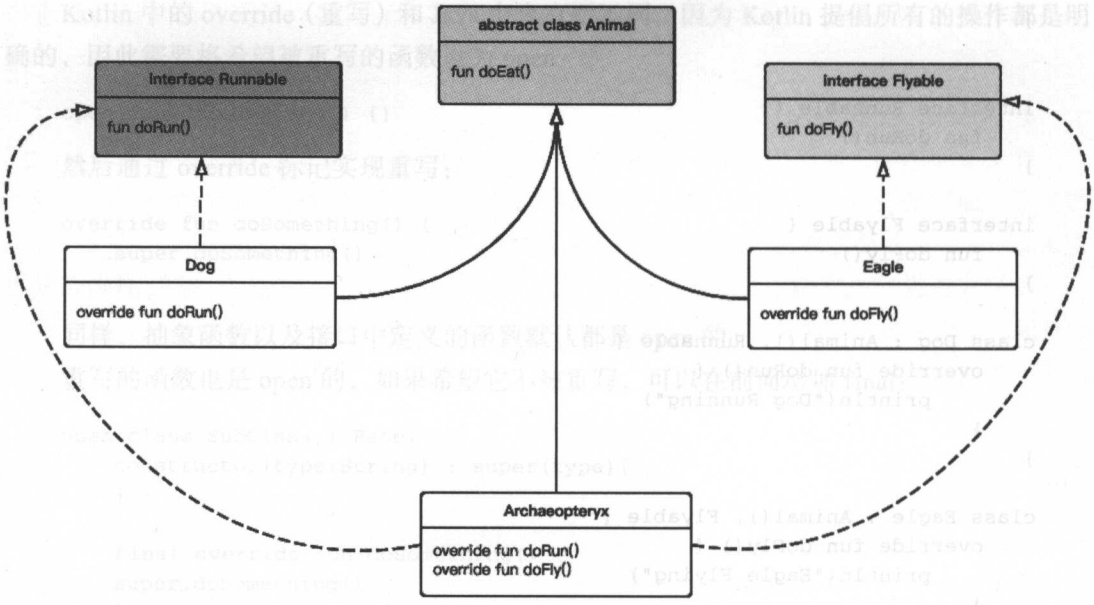


图 7-3 上述代码类之间的关系

我们可以看出，Archaeopteryx 继承了 Animal 类，用了父类 doEat() 函数功能；实现了 Runnable 接口，拥有了 doRun() 函数规范；实现了 Flyable 接口，拥有了 doFly() 函数规范。

在这里，我们通过实现多个接口，组合完成了的多个功能，而不是设计多个层次的复杂的继承关系。

7.7 枚举类

Kotlin 的枚举类定义如下：

```
public abstract class Enum<E : Enum<E>>(name: String, ordinal: Int): Comparable<E> {
    companion object {}

    public final val name: String
    public final val ordinal: Int

    public override final fun compareTo(other: E): Int
    protected final fun clone(): Any

    public override final fun equals(other: Any?): Boolean
    public override final fun hashCode(): Int
    public override fun toString(): String
}
```

我们可以看出，这个枚举类有两个属性：

```
public final val name: String
public final val ordinal: Int
```

分别表示的是枚举对象的值与下标位置。

同时，我们可以看出枚举类还实现了 Comparable 接口。

7.7.1　枚举类基本用法

枚举类的最基本的用法是实现类型安全的枚举：

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}

>>> val north = Direction.NORTH
>>> north.name
NORTH
>>> north.ordinal
0
>>> north is Direction
true
```

每个枚举常量都是一个对象。枚举常量用逗号分隔。

7.7.2　初始化枚举值

我们可以如下初始化枚举类的值：

```
enum class Color(val rgb: Int) {
        RED(0xFF0000),
        GREEN(0x00FF00),
        BLUE(0x0000FF)
}

>>> val red = Color.RED
>>> red.rgb
16711680
```

另外，枚举常量也可以声明自己的匿名类：

```
enum class ActivtyLifeState {
    onCreate {
        override fun signal() = onStart
    },

    onStart {
        override fun signal() = onStop
    },

    onStop {
```

```
        override fun signal() = onStart
    },

    onDestroy {
        override fun signal() = onDestroy
    };

    abstract fun signal(): ActivtyLifeState
}

>>> val s = ActivtyLifeState.onCreate
>>> println(s.signal())
onStart
```

7.7.3 使用枚举常量

我们使用 enumValues() 函数来列出枚举的所有值：

```
@SinceKotlin("1.1")
public inline fun <reified T : Enum<T>> enumValues(): Array<T>
```

每个枚举常量，默认都有 name（名称）和 ordinal（位置）的属性（这与 Java 的 Enum 类里面的类似）：

```
val name: String
val ordinal: Int
```

代码示例：

```
enum class RGB { RED, GREEN, BLUE }

>>> val rgbs = enumValues<RGB>().joinToString { "${it.name} : ${it.ordinal} " }
>>> rgbs
RED : 0 , GREEN : 1 , BLUE : 2
```

我们直接声明了一个简单枚举类，使用遍历函数 enumValues<RGB>() 列出了 RGB 枚举类的所有枚举值。使用 it.name it.ordinal 直接访问各个枚举值的名称和位置。

另外，我们也可以自定义枚举属性值：

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}

>>> val colors = enumValues<Color>().joinToString { "${it.rgb} : ${it.name} :
    ${it.ordinal} " }
>>> colors
16711680 : RED : 0 , 65280 : GREEN : 1 , 255 : BLUE : 2
```

然后，可以直接使用 it.rgb 访问属性名来得到对应的属性值。

7.8　注解类

Kotlin 的注解与 Java 的注解完全兼容。

7.8.1　声明注解

Annotation class（注解名）的代码示例如下：

```
@Target(AnnotationTarget.CLASS,
        AnnotationTarget.FUNCTION,
        AnnotationTarget.EXPRESSION,
        AnnotationTarget.FIELD,
        AnnotationTarget.LOCAL_VARIABLE,
        AnnotationTarget.TYPE,
        AnnotationTarget.TYPEALIAS,
        AnnotationTarget.TYPE_PARAMETER,
        AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
@Repeatable
annotation class MagicClass


@Target(AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
@Repeatable
annotation class MagicFunction


@Target(AnnotationTarget.CONSTRUCTOR)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
@Repeatable
annotation class MagicConstructor
```

在上面的代码中，通过向注解类添加元注解（meta-annotation）的方法来指定其他属性：

- @Target：指定这个注解可用于哪些元素（类、函数、属性、表达式，等等）。
- @Retention：指定这个注解的信息是否被保存到编译后的 class 文件中，以及在运行时是否可以通过反 射访问到它。
- @Repeatable：允许在单个元素上多次使用同一个注解。
- @MustBeDocumented：表示这个注解是公开 API 的一部分，在自动产生的 API 文档的类或者函数签名中，应该包含这个注解的信息。

这几个注解定义在 kotlin/annotation/Annotations.kt 类中。

7.8.2 使用注解

注解可以用在类、函数、参数、变量（成员变量、局部变量）、表达式、类型上，这由该注解的元注解 @Target 定义：

```
@MagicClass class Foo @MagicConstructor constructor() {

    constructor(index: Int) : this() {
        this.index = index
    }

    @MagicClass var index: Int = 0
    @MagicFunction fun magic(@MagicClass name: String) {

    }
}
```

注解在主构造器上，主构造器必须加上关键字“constructor”：

```
@MagicClass class Foo @MagicConstructor constructor() {
...
}
```

7.9 单例模式与伴生对象

7.9.1 单例模式

单例模式（Singleton）很常用，是一种常见的软件设计模式，例如，Spring 中的 Bean 默认就是单例。通过单例模式可以保证系统中一个类只有一个实例，即一个类只有一个对象实例。

我们用 Java 实现一个简单的单例类的代码如下：

```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

测试代码：

```
Singleton singleton1 = Singleton.getInstance();
```

可以看出，我们先在单例类中声明了一个私有静态的 Singleton instance 变量，然后声明一个私有构造函数 private Singleton() {}，这个私有构造函数使得外部无法直接通过 new 的方式来构建对象：

```
Singleton singleton2 = new Singleton(); //error, cannot private access
```

最后提供一个 public 静态方法 getInstance() 来获取当前类的唯一实例的。这里给出的是一个简单的单例类，是线程不安全的。

7.9.2　object 对象

Kotlin 中没有“静态属性和方法”，但是也提供了实现类似于“单例”的功能，我们可以使用关键字 object 声明一个 object 对象：

```
object AdminUser {
    val username: String = "admin"
    val password: String = "admin"
    fun getTimestamp() = SimpleDateFormat("yyyyMMddHHmmss").format(Date())
    fun md5Password() = EncoderByMd5(password + getTimestamp())
}
```

测试代码：

```
val adminUser = AdminUser.username
val adminPassword = AdminUser.md5Password()
println(adminUser)  //admin
println(adminPassword)  //g+0yLfaPVYxUf6TMIdXFXw==, 这个值具体运行时会变化
```

为了方便在 REPL 中演示说明，我们再写一个示例代码：

```
>>> object User {
...     val username: String = "admin"
...     val password: String = "admin"
... }
```

object 对象只能通过对象名字来访问：

```
>>> User.username
admin
>>> User.password
admin
```

不能像下面这样使用构造函数：

```
>>> val u = User()
error: expression 'User' of type 'Line130.User' cannot be invoked as a function.
    The function 'invoke()' is not found
val u = User()
        ^
```

为了更加直观地了解 object 对象的概念，我们把上面的 object User 的代码反编译成 Java 代码：

```
public final class User {
   @NotNull
   private static final String username = "admin";
   @NotNull
   private static final String password = "admin";
   public static final User INSTANCE;

   @NotNull
   public final String getUsername() {
      return username;
   }

   @NotNull
   public final String getPassword() {
      return password;
   }

   private User() {
      INSTANCE = (User)this;
      username = "admin";
      password = "admin";
   }

   static {
      new User();
   }
}
```

从上面的反编译代码，我们可以直观了解 Kotlin 的 object 背后的一些原理。

7.9.3 嵌套 object 对象

这个 object 对象还可以放到一个类里面：

```
class DataProcessor {
    fun process() {
        println("Process Data")
    }


    object FileUtils {
        val userHome = "/Users/jack/"

        fun getFileContent(file: String): String {
            var content = ""
            val f = File(file)
            f.forEachLine { content = content + it + "\n" }
```

```
            return content
        }

    }
}
```

测试代码：

```
DataProcessor.FileUtils.userHome // /Users/jack/
DataProcessor.FileUtils.getFileContent("test.data") // 输出文件的内容
```

同样，我们只能通过类的名称来直接访问 object，不能使用对象实例引用。下面的写法是错误的：

```
val dp = DataProcessor()
dp.FileUtils.userHome // error, Nested object FileUtils cannot access object
    via reference
```

我们在 Java 中通常会写一些 Utils 类，这样的类我们在 Kotlin 中就可以直接使用 object 对象：

```
object HttpUtils {
    val client = OkHttpClient()

    @Throws(Exception::class)
    fun getSync(url: String): String? {
        val request = Request.Builder()
                .url(url)
                .build()

        val response = client.newCall(request).execute()
        if (!response.isSuccessful()) throw IOException("Unexpected code " + response)

        val responseHeaders = response.headers()
        for (i in 0..responseHeaders.size() - 1) {
            println(responseHeaders.name(i) + ": " + responseHeaders.value(i))
        }
        return response.body()?.string()
    }

    @Throws(Exception::class)
    fun getAsync(url: String) {
        var result: String? = ""

        val request = Request.Builder()
                .url(url)
                .build()
        client.newCall(request).enqueue(object : Callback {
            override fun onFailure(call: Call, e: IOException?) {
                e?.printStackTrace()
```

```
            }

            @Throws(IOException::class)
            override fun onResponse(call: Call, response: Response) {
                if (!response.isSuccessful()) throw IOException("Unexpected
                    code " + response)

                val responseHeaders = response.headers()
                for (i in 0..responseHeaders.size() - 1) {
                    println(responseHeaders.name(i) + ": " + responseHeaders.value(i))
                }
                result = response.body()?.string()
                println(result)

            }
        })
    }
}
```

测试代码：

```
val url = "http://www.baidu.com"
val html1 = HttpUtils.getSync(url)      // 同步 get
println("html1=${html1}")
HttpUtils.getAsync(url)                 // 异步 get
```

7.9.4 匿名 object

在代码行内，有时候需要的仅仅是一个简单的对象，这个时候就可以使用下面的匿名 object 的方式：

```
fun distance(x: Double, y: Double): Double {
    val porigin = object {
        var x = 0.0
        var y = 0.0
    }
    return Math.sqrt((x - porigin.x) * (x - porigin.x) + (y - porigin.y) * (y - porigin.y))
}
```

测试代码：

```
distance(3.0, 4.0)
```

需要注意的是，匿名对象只可以用在本地和私有作用域中声明的类型。代码示例：

```
class AnonymousObjectType {
    // 私有函数，返回的是匿名 object 类型
    private fun privateFoo() = object {
        val x: String = "x"
    }

    // 公有函数，返回的类型是 Any
```

```
    fun publicFoo() = object {
        val x: String = "x" //无法访问到
    }

    fun test() {
        val x1 = privateFoo().x        //Works
        //val x2 = publicFoo().x  //ERROR: Unresolved reference 'x'
    }
}


fun main(args: Array<String>) {
    AnonymousObjectType().publicFoo().x //Unresolved reference 'x'
}
```

与 Java 匿名内部类类似，object 对象表达式中的代码可以访问来自包含它的作用域的变量，与 Java 不同的是，这不限于 final 变量：

```
fun countCompare() {
    var list = mutableListOf(1, 4, 3, 7, 11, 9, 10, 20)
    var countCompare = 0
    Collections.sort(list, object : Comparator<Int> {
        override fun compare(o1: Int, o2: Int): Int {
            countCompare++
            println("countCompare=$countCompare")
            println(list)
            return o1.compareTo(o2)
        }
    })
}
```

测试代码：

```
countCompare()

countCompare=1
[1, 4, 3, 7, 11, 9, 10, 20]
...
countCompare=17
[1, 3, 4, 7, 9, 10, 11, 20]
```

7.9.5　伴生对象

Kotlin 中还提供了“伴生对象”(companion object)，用 companion object 关键字声明：

```
class DataProcessor {
    fun process() {
        println("Process Data")
    }

    object FileUtils {
```

```
        val userHome = "/Users/jack/"

        fun getFileContent(file: String): String {
            var content = ""
            val f = File(file)
            f.forEachLine { content = content + it + "\n" }
            return content
        }

    }

    companion object StringUtils {
        fun isEmpty(s: String): Boolean {
            return s.isEmpty()
        }
    }

}
```

一个类只能有 1 个伴生对象。也就是是下面的写法是错误的：

```
class ClassA {
    companion object Factory {
        fun create(): ClassA = ClassA()
    }

    companion object Factory2 { //error, only 1 companion object is allowed per class
        fun create(): MyClass = MyClass()
    }
}
```

一个类的伴生对象默认引用名是 Companion：

```
class ClassB {
    companion object {
        fun create(): ClassB = ClassB()
        fun get() = "Hi, I am CompanyB"
    }
}
```

我们可以直接像在 Java 静态类中使用静态方法一样使用一个类的伴生对象的函数和属性：

```
ClassB.Companion.index
ClassB.Companion.create()
ClassB.Companion.get()
```

其中，Companion 可以省略不写：

```
ClassB.index
ClassB.create()
ClassB.get()
```

当然，我们也可以指定伴生对象的名称：

```
class ClassC {
    var index = 0
    fun get(index: Int): Int {
        return 0
    }

    companion object CompanyC {
        fun create(): ClassC = ClassC()
        fun get() = "Hi, I am CompanyC"
    }
}
```

测试代码：

```
ClassC.index
ClassC.create()//com.easy.kotli.ClassC@7440e464，具体运行值会变化
ClassC.get() //Hi, I am CompanyC
ClassC.CompanyC.index
ClassC.CompanyC.create()
ClassC.CompanyC.get()
```

伴生对象的初始化是在相应的类被加载解析时，与 Java 静态初始化器的语义相匹配。

即使伴生对象的成员看起来像其他语言的静态成员，在运行时仍然是真实对象的实例成员。而且，还可以实现接口：

```
interface BeanFactory<T> {
    fun create(): T
}


class MyClass {
    companion object : BeanFactory<MyClass> {
        override fun create(): MyClass {
            println("MyClass Created!")
            return MyClass()
        }
    }
}
```

测试代码：

```
MyClass.create()  //"MyClass Created!"
MyClass.Companion.create() //"MyClass Created!"
```

另外，如果想使用 Java 中的静态成员和静态方法的话，我们可以用如下方法：

- @JvmField 注解：生成与该属性相同的静态字段。
- @JvmStatic 注解：在单例对象和伴生对象中生成对应的静态方法。

7.10 密封类

就像我们为什么要用 enum 类型一样，比如你有一个 enum 类型 MoneyUnit，定义了元、角、分这些单位。枚举就是为了控制住你所有要的情况是正确的，而不是用硬编码方式写成字符串“元”，“角”，“分”。

同样，密封类（sealed）的目的类似，一个类之所以设计成 sealed，就是为了限制类的继承结构，将一个值限制在有限集中的类型中，而不能有任何其他的类型。

在某种意义上，sealed 类是枚举类的扩展：枚举类型的值集合也是受限的，但每个枚举常量只存在一个实例，而密封类的一个子类可以有可包含状态的多个实例。

要声明一个密封类，需要在类名前面添加 sealed 修饰符。密封类的所有子类都必须与密封类在同一个文件中声明（在 Kotlin 1.1 之前，该规则更加严格：子类必须嵌套在密封类声明的内部）：

```
sealed class Expression

class Unit : Expression()
data class Const(val number: Double) : Expression()
data class Sum(val e1: Expression, val e2: Expression) : Expression()
data class Multiply(val e1: Expression, val e2: Expression) : Expression()
object NaN : Expression()
```

使用密封类的主要场景是在使用 when 表达式的时候，能够验证语句覆盖了所有情况，而无需再添加一个 else 子句：

```
fun eval(expr: Expression): Double = when (expr) {
    is Unit -> 1.0
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    is Multiply -> eval(expr.e1) * eval(expr.e2)
    NaN -> Double.NaN
// 不再需要 `else` 子句，因为我们已经覆盖了所有的情况
}
```

测试代码：

```
fun main(args: Array<String>) {
    val u = eval(Unit())
    val a = eval(Const(1.1))
    val b = eval(Sum(Const(1.0), Const(9.0)))
    val c = eval(Multiply(Const(10.0), Const(10.0)))
    println(u)
    println(a)
    println(b)
    println(c)
}
```

输出：

```
1.0
1.1
10.0
100.0
```

7.11　数据类

7.11.1　构造函数中的 val/var

在开始讲数据类之前，我们先来看一下几种类声明的写法。

写法一：

```
class Aook(name: String)
```

这样写，这个 name 变量是无法被外部访问到的。它对应的反编译之后的 Java 代码如下：

```
public final class Aook {
    public Aook(@NotNull String name) {
        Intrinsics.checkParameterIsNotNull(name, "name");
        super();
    }
}
```

写法二：要想这个 name 变量被访问到，我们可以在类体中再声明一个变量，然后把这个构造函数中的参数赋值给它：

```
class Cook(name: String) {
    val name = name
}
```

测试代码：

```
val cook = Cook("Cook")
cook.name
```

对应的 Java 实现代码是：

```
public final class Cook {
    @NotNull
    private final String name;

    @NotNull
    public final String getName() {
        return this.name;
    }

    public Cook(@NotNull String name) {
```

```
        Intrinsics.checkParameterIsNotNull(name, "name");
        super();
        this.name = name;
    }
}
```

写法三：

```
class Dook(val name: String)
class Eook(var name: String)
```

构造函数中带 var、val 修饰的变量，Kotlin 编译器会自动为它们生成 getter、setter 函数。上面的写法对应的 Java 代码就是：

```
public final class Dook {
    @NotNull
    private final String name;

    @NotNull
    public final String getName() {
        return this.name;
    }

    public Dook(@NotNull String name) {
        Intrinsics.checkParameterIsNotNull(name, "name");
        super();
        this.name = name;
    }
}

public final class Eook {
    @NotNull
    private String name;

    @NotNull
    public final String getName() {
        return this.name;
    }

    public final void setName(@NotNull String var1) {
        Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
        this.name = var1;
    }

    public Eook(@NotNull String name) {
        Intrinsics.checkParameterIsNotNull(name, "name");
        super();
        this.name = name;
    }
}
```

测试代码：

```
val dook = Dook("Dook")
dook.name
val eook = Eook("Eook")
eook.name
```

7.11.2　领域实体类

我们写 Java 代码的时候，会经常创建一些只保存数据的类。比如说：

- POJO 类：POJO 全称是 Plain Ordinary Java Object/Pure Old Java Object，中文可以翻译成：普通 Java 类，具有一部分 getter/setter 方法的那种类就可以称作 POJO。
- DTO 类：Data Transfer Object，数据传输对象类，泛指用于展示层与服务层之间的数据传输对象。
- VO 类：VO 有两种说法，一个是 ViewObject，一个是 ValueObject。
- PO 类：Persisent Object，持久对象。它们是由一组属性和属性的 get 和 set 方法组成。PO 是在持久层所使用，用来封装原始数据。
- BO 类：Business Object，业务对象层，表示应用程序领域内“事物”的所有实体类。
- DO 类：Domain Object，领域对象，就是从现实世界中抽象出来的有形或无形的业务实体。

还有许多。这些我们统称为领域模型中的实体类。最简单的实体类是 POJO 类，含有属性及属性对应的 get 和 set 方法，实体类常见的方法还有用于输出自身数据的 toString 方法。

7.11.3　数据类的概念

在 Kotlin 中，也有对应这样的领域实体类的概念，并在语言层面上做了支持，叫做数据类（data class）：

```
data class Book(val name: String)
data class Fook(var name: String)
data class User(
        val name: String,
        val gender: String,
        val age: Int
) {
    fun validate(): Boolean {
        return true
    }
}
```

这里的 var/val 是必须要带上的。因为编译器要把主构造函数中声明的所有属性，自动生成以下函数：

```
equals()/hashCode()
toString() : 格式是 User(name=Jacky, gender=Male, age=10)
```

componentN() 函数 ：按声明顺序对应于所有属性 component1()、component2() ...
copy() 函数

如果我们自定义了这些函数，或者继承父类重写了这些函数，编译器就不会再去生成。测试代码：

```
val book = Book("Book")
book.name
book.copy("Book2")

val jack = User("Jack", "Male", 1)
jack.name
jack.gender
jack.age
jack.toString()
jack.validate()


val olderJack = jack.copy(age = 2)
val anotherJack = jack.copy(name = "Jacky", age = 10)
```

在一些场景下，我们需要复制一个对象来改变它的部分属性，而其余部分保持不变。copy() 函数就是为此而生成。例如上面的的 User 类的 copy 函数的使用：

```
val olderJack = jack.copy(age = 2)
val anotherJack = jack.copy(name = "Jacky", age = 10)
```

7.11.4 数据类的限制

数据类有以下的限制要求：

1）主构造函数需要至少有一个参数。下面的写法是错误的：

```
data class Gook //error, data class must have at least one primary constructor parameter
```

2）主构造函数的所有参数需要标记为 val 或 var；

```
data class Hook(name: String)//error, data class must have only var/val property
```

与普通类一样，数据类也可以有次级构造函数：

```
data class LoginUser(val name: String = "", val password: String = "") : DBase(),
    IBaseA, IBaseB {

    var isActive = true

    constructor(name: String, password: String, isActive: Boolean) : this(name,
        password) {
        this.isActive = isActive
    }
    ...
}
```

3）数据类不能是抽象、开放、密封或者内部的，也就是说，下面的写法都是错误的：

```
abstract data class Iook(val name: String) //modifier abstract is incompatible with data
open data class Jook(val name: String)     //modifier abstract is incompatible with data
sealed data class Kook(val name: String)   //modifier sealed is incompatible with data
inner data class Look(val name: String)    //modifier inner is incompatible with data
```

数据类只能是 final 的：

```
final data class Mook(val name: String) //modifier abstract is incompatible
    with data
```

4）在 1.1 之前数据类只能实现接口，自 1.1 起，数据类可以扩展其他类。代码示例：

```
open class DBase
interface IBaseA
interface IBaseB

data class LoginUser(val name: String, val password: String) : DBase(), IBaseA,
    IBaseB {

    override fun equals(other: Any?): Boolean {
        return super.equals(other)
    }

    override fun hashCode(): Int {
        return super.hashCode()
    }

    override fun toString(): String {
        return super.toString()
    }

    fun validate(): Boolean {
        return true
    }
}
```

测试代码：

```
val loginUser1 = LoginUser("Admin", "admin")
println(loginUser1.component1())
println(loginUser1.component2())
println(loginUser1.name)
println(loginUser1.password)
println(loginUser1.toString())
```

输出：

```
Admin
admin
Admin
```

```
admin
com.easy.kotlin.LoginUser@7440e464
```

可以看出，由于我们重写了 override fun toString(): String，对应的输出使我们熟悉的类的输出格式。

如果我们不重写这个 toString 函数，则会默认输出：

```
LoginUser(name=Admin, password=admin)
```

上面的类声明的构造函数，要求我们每次必须初始化 name、password 的值，如果我们想拥有一个无参的构造函数，我们只要对所有的属性指定默认值即可：

```
data class LoginUser(val name: String = "", val password: String = "") : DBase(),
    IBaseA, IBaseB {
...
}
```

这样我们在创建对象的时候，就可以直接使用：

```
val loginUser3 = LoginUser()
loginUser3.name
loginUser3.password
```

7.11.5 数据类的解构

解构相当于 Component 函数的逆向映射：

```
val helen = User("Helen", "Female", 15)
val (name, gender, age) = helen
println("$name, $gender, $age years of age")
```

输出：

```
Helen, Female, 15 years of age
```

7.11.6 标准数据类 Pair 和 Triple

标准库中的二元组 Pair 类就是一个数据类：

```
public data class Pair<out A, out B>(
        public val first: A,
        public val second: B) : Serializable {
    public override fun toString(): String = "($first, $second)"
}
```

Kotlin 标准库中，对 Pair 类还增加了转换成 List 的扩展函数：

```
public fun <T> Pair<T, T>.toList(): List<T> = listOf(first, second)
```

还有三元组 Triple 类：

```
public data class Triple<out A, out B, out C>(
```

```
        public val first: A,
        public val second: B,
        public val third: C) : Serializable {
    public override fun toString(): String = "($first, $second, $third)"
}
fun <T> Triple<T, T, T>.toList(): List<T> = listOf(first, second, third)
```

7.12　嵌套类

7.12.1　嵌套类：类中的类

类可以嵌套在其他类中，可以嵌套多层：

```
class NestedClassesDemo {
    class Outer {
        private val zero: Int = 0
        val one: Int = 1

        class Nested {
            fun getTwo() = 2
            class Nested1 {
                val three = 3
                fun getFour() = 4
            }
        }
    }
}
```

测试代码：

```
val one = NestedClassesDemo.Outer().one
val two = NestedClassesDemo.Outer.Nested().getTwo()
val three = NestedClassesDemo.Outer.Nested.Nested1().three
val four = NestedClassesDemo.Outer.Nested.Nested1().getFour()
println(one)
println(two)
println(three)
println(four)
```

我们可以看出，访问嵌套类（Nested Class）的方式是直接使用“类名.”，有多少层嵌套，就用多少层类名加点号来访问。

普通的嵌套类，没有持有外部类的引用，所以是无法访问外部类的变量的：

```
class NestedClassesDemo {
class Outer {
        private val zero: Int = 0
        val one: Int = 1


        class Nested {
```

```
            fun getTwo() = 2

            fun accessOuter() = {
                println(zero) //error, cannot access outer class
                println(one)  //error, cannot access outer class
            }
        }
}
}
```

我们在 Nested 类中，访问不到 Outer 类中的变量 zero，one。

如果想要访问到，我们只需要在 Nested 类前面加上 inner 关键字修饰，表明这是一个嵌套的内部类。

7.12.2 内部类

类可以标记为 inner 以便能够访问外部类的成员，这样的类叫内部类（Inner Class）。内部类会带有一个对外部类的对象的引用：

```
class NestedClassesDemo {
class Outer {
        private val zero: Int = 0
        val one: Int = 1

        inner class Inner {
            fun accessOuter() = {
                println(zero) //works
                println(one)  //works
            }

        }
}
```

测试代码：

```
val innerClass = NestedClassesDemo.Outer().Inner().accessOuter()
```

我们可以看到，当访问 inner class Inner 的时候，使用的是 Outer().Inner()，这是持有了 Outer 的对象引用，与普通嵌套类直接使用类名访问的方式区分。

7.12.3 匿名内部类

匿名内部类（Annonymous Inner Class）就是没有名字的内部类。既然是内部类，那么它自然也可以访问外部类的变量。

我们使用对象表达式创建一个匿名内部类实例：

```
class NestedClassesDemo {
class AnonymousInnerClassDemo {
```

```
        var isRunning = false
        fun doRun() {
            Thread(object : Runnable {
                override fun run() {
                    isRunning = true
                    println("doRun : i am running, isRunning = $isRunning")
                }
            }).start()
        }
}
}
```

如果对象是函数式 Java 接口，即具有单个抽象方法的 Java 接口的实例，例如上面的例子中的 Runnable 接口：

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

我们可以使用 lambda 表达式创建它，下面的几种写法都是可以的：

```
fun doStop() {
    var isRunning = true
    Thread({
        isRunning = false
        println("doStop: i am not running, isRunning = $isRunning")
    }).start()
}

fun doWait() {
    var isRunning = true

    val wait = Runnable {
        isRunning = false
        println("doWait: i am waiting, isRunning = $isRunning")
    }

    Thread(wait).start()
}

fun doNotify() {
    var isRunning = true

    val wait = {
        isRunning = false
        println("doNotify: i notify, isRunning = $isRunning")
    }

    Thread(wait).start()
}
```

测试代码：

```
NestedClassesDemo.Outer.AnonymousInnerClassDemo().doRun()
NestedClassesDemo.Outer.AnonymousInnerClassDemo().doStop()
NestedClassesDemo.Outer.AnonymousInnerClassDemo().doWait()
NestedClassesDemo.Outer.AnonymousInnerClassDemo().doNotify()
```

输出：

```
doRun : i am running, isRunning = true
doStop: i am not running, isRunning = false
doWait: i am waiting, isRunning = false
doNotify: i notify, isRunning = false
```

关于 lambda 表达式以及函数式编程，我们将在下一章中学习。

7.13 委托

7.13.1 代理模式

代理模式（Proxy Pattern）也称委托模式。在代理模式中，有两个对象参与处理同一个请求，接受请求的对象将请求委托（Delegation）给另一个对象来处理。代理模式是一项基本技巧，许多其他的模式，如状态模式、策略模式、访问者模式本质上是在特殊的场合采用了代理模式。

代理模式使得我们可以用聚合来替代继承，它还使我们可以模拟 mixin（混合类型）。委托模式的作用是将委托者与实际实现代码分离出来，以达成解耦的目的。

一个代理模式的 Java 代码示例：

```
package com.easy.kotlin;

/**
 * Created by jack on 2017/7/5.
 */
interface JSubject {
    public void request();
}

class JRealSubject implements JSubject {
    @Override
    public void request() {
        System.out.println("JRealSubject Requesting");
    }
}

class JProxy implements JSubject {
    private JSubject subject = null;

    //通过构造函数传递代理者
```

```
    public JProxy(JSubject sub) {
        this.subject = sub;
    }

    @Override
    public void request() { //实现接口中定义的方法
        this.before();
        this.subject.request();
        this.after();
    }

    private void before() {
        System.out.println("JProxy Before Requesting ");
    }

    private void after() {
        System.out.println("JProxy After Requesting ");
    }
}

public class DelegateDemo {
    public static void main(String[] args) {
        JRealSubject jRealSubject = new JRealSubject();
        JProxy jProxy = new JProxy(jRealSubject);
        jProxy.request();
    }
}
```

输出：

```
JProxy Before Requesting
JRealSubject Requesting
JProxy After Requesting
```

7.13.2　类的委托

就像支持单例模式的 object 对象一样，Kotlin 在语言层面原生支持委托模式。我们来看一个类的委托（Class Delegation）的例子。

代码示例：

```
package com.easy.kotlin

import java.util.*

/**
 * Created by jack on 2017/7/5.
 */

interface Subject {
    fun hello()
```

```
}

class RealSubject(val name: String) : Subject {
    override fun hello() {
        val now = Date()
        println("Hello, REAL $name! Now is $now")
    }
}

class ProxySubject(val sb: Subject) : Subject by sb {
    override fun hello() {
        println("Before ! Now is ${Date()}")
        sb.hello()
        println("After ! Now is ${Date()}")
    }
}

fun main(args: Array<String>) {
    val subject = RealSubject("World")
    subject.hello()
    println("-------------------------")
    val proxySubject = ProxySubject(subject)
    proxySubject.hello()
}
```

在这个例子中，委托代理类 ProxySubject 继承接口 Subject，并将其所有共有的方法委托给一个指定的对象 sb：

```
class ProxySubject(val sb: Subject) : Subject by sb
```

ProxySubject 的超类型 Subject 中的 by sb 表示 sb 将会在 ProxySubject 中内部存储。另外，我们在覆盖重写了函数 override fun hello()。

测试代码：

```
fun main(args: Array<String>) {
    val subject = RealSubject("World")
    subject.hello()
    println("-------------------------")
    val proxySubject = ProxySubject(subject)
    proxySubject.hello()
}
```

输出：

```
Hello, REAL World! Now is Wed Jul 05 02:45:42 CST 2017
-------------------------
Before ! Now is Wed Jul 05 02:45:42 CST 2017
Hello, REAL World! Now is Wed Jul 05 02:45:42 CST 2017
After ! Now is Wed Jul 05 02:45:42 CST 2017
```

7.13.3　委托属性

通常对于属性类型，我们是在每次需要的时候手动声明它们：

```
class NormalPropertiesDemo {
    var content: String = "NormalProperties init content"
}
```

那么这个 content 属性将会很“呆板”。委托属性（Delegated Properties）赋予了属性富有变化的活力。

例如：

- 延迟属性（lazy properties）：其值只在首次访问时计算。
- 可观察属性（observable properties）：监听器会收到有关此属性变更的通知。
- 把多个属性储存在一个映射（map）中，而不是每个存在单独的字段中。

1. Kotlin 的委托属性

Kotlin 支持委托属性：

```
class DelegatePropertiesDemo {
    var content: String by Content()

    override fun toString(): String {
        return "DelegatePropertiesDemo Class"
    }
}

class Content {
    operator fun getValue(delegatePropertiesDemo: DelegatePropertiesDemo,
        property: KProperty<*>): String {
        return "${delegatePropertiesDemo} property '${property.name}' =
            'Balalala ... ' "
    }

    operator fun setValue(delegatePropertiesDemo: DelegatePropertiesDemo,
        property: KProperty<*>, value: String) {
        println("${delegatePropertiesDemo} property '${property.name}' is
            setting value: '$value'")
    }
}
```

在 var content: String by Content() 中，by 后面的表达式的 Content() 就是该属性“委托”的对象。content 属性对应的 get()（和 set()）会被委托给 Content() 的 operator fun getValue() 和 operator fun setValue() 函数，这两个函数是必须的，而且得是操作符函数。

测试代码：

```
val n = NormalPropertiesDemo()
println(n.content)
```

```
n.content = "Lao tze"
println(n.content)

val e = DelegatePropertiesDemo()
println(e.content) //call Content.getValue
e.content = "Confucius" //call Content.setValue
println(e.content) //call Content.getValue
```

输出：

```
NormalProperties init content
Lao tze
DelegatePropertiesDemo Class property 'content' = 'Balalala ... '
DelegatePropertiesDemo Class property 'content' is setting value: 'Confucius'
DelegatePropertiesDemo Class property 'content' = 'Balalala ...
```

2. 懒加载属性委托（ lazy）

lazy() 函数定义如下：

```
@kotlin.jvm.JvmVersion
public fun <T> lazy(initializer: () -> T): Lazy<T> = SynchronizedLazyImpl(initializer)
```

它接受一个 lambda 并返回一个 Lazy 实例的函数，返回的实例可以作为实现懒加载属性的委托：

第一次调用 get() 会执行已传递给 lazy() 的 lamda 表达式并记录下结果，后续调用 get() 只是返回之前记录的结果。

代码示例：

```
val synchronizedLazyImpl = lazy({
    println("lazyValueSynchronized1  3!")
    println("lazyValueSynchronized1  2!")
    println("lazyValueSynchronized1  1!")
    "Hello, lazyValueSynchronized1 ! "
})

val lazyValueSynchronized1: String by synchronizedLazyImpl
println(lazyValueSynchronized1)
println(lazyValueSynchronized1)

val lazyValueSynchronized2: String by lazy {
    println("lazyValueSynchronized2  3!")
    println("lazyValueSynchronized2  2!")
    println("lazyValueSynchronized2  1!")
    "Hello, lazyValueSynchronized2 ! "
}

println(lazyValueSynchronized2)
println(lazyValueSynchronized2)
```

输出：

```
lazyValueSynchronized1  3!
lazyValueSynchronized1  2!
lazyValueSynchronized1  1!
Hello, lazyValueSynchronized1 !
Hello, lazyValueSynchronized1 !


lazyValueSynchronized2  3!
lazyValueSynchronized2  2!
lazyValueSynchronized2  1!
Hello, lazyValueSynchronized2 !
Hello, lazyValueSynchronized2 !
```

默认情况下，对于 lazy 属性的求值是同步的（synchronized），下面两种写法是等价的：

```
val synchronizedLazyImpl = lazy({
    println("lazyValueSynchronized1  3!")
    println("lazyValueSynchronized1  2!")
    println("lazyValueSynchronized1  1!")
    "Hello, lazyValueSynchronized1 ! "
})

val synchronizedLazyImpl2 = lazy(LazyThreadSafetyMode.SYNCHRONIZED, {
    println("lazyValueSynchronized1  3!")
    println("lazyValueSynchronized1  2!")
    println("lazyValueSynchronized1  1!")
    "Hello, lazyValueSynchronized1 ! "
})
```

该值是线程安全的。所有线程会看到相同的值。

如果初始化委托多个线程可以同时执行，不需要同步锁，使用 LazyThreadSafetyMode.PUBLICATION：

```
val lazyValuePublication: String by lazy(LazyThreadSafetyMode.PUBLICATION, {
    println("lazyValuePublication 3!")
    println("lazyValuePublication 2!")
    println("lazyValuePublication 1!")
    "Hello, lazyValuePublication ! "
})
```

而如果属性的初始化是单线程的，那么我们使用 LazyThreadSafetyMode.NONE 模式（性能最高）：

```
val lazyValueNone: String by lazy(LazyThreadSafetyMode.NONE, {
    println("lazyValueNone 3!")
    println("lazyValueNone 2!")
    println("lazyValueNone 1!")
    "Hello, lazyValueNone ! "
})
```

3. 可观察属性委托（Delegates. observable）

我们把属性委托给 Delegates.observable 函数，当属性值被重新赋值的时候，触发其中的回调函数 onChange。

该函数定义如下：

```
public inline fun <T> observable(initialValue: T, crossinline onChange:
    (property: KProperty<*>, oldValue: T, newValue: T) -> Unit):
        ReadWriteProperty<Any?, T> = object : ObservableProperty<T>(initialValue) {
            override fun afterChange(property: KProperty<*>, oldValue: T, newValue:
                T) = onChange(property, oldValue, newValue)
        }
```

代码示例：

```
class PostHierarchy {
    var level: String by Delegates.observable("P0",
            { property: KProperty<*>,
              oldValue: String,
              newValue: String ->
                println("$oldValue -> $newValue")
            })
}
```

测试代码：

```
val ph = PostHierarchy()
ph.level = "P1"
ph.level = "P2"
ph.level = "P3"
println(ph.level) // P3
```

输出：

```
P0 -> P1
P1 -> P2
P2 -> P3
P3
```

我们可以看出，属性 level 每次赋值，都回调了 Delegates.observable 中的 lambda 表达式所写的 onChange 函数。

4. 可否决属性委托（Delegates. vetoable）

这个函数定义如下：

```
public inline fun <T> vetoable(initialValue: T, crossinline onChange: (property:
    KProperty<*>, oldValue: T, newValue: T) -> Boolean):
        ReadWriteProperty<Any?, T> = object : ObservableProperty<T>(initialValue) {
            override fun beforeChange(property: KProperty<*>, oldValue: T,
                newValue: T): Boolean = onChange(property, oldValue, newValue)
        }
```

当我们把属性委托给这个函数时，可以通过 onChange 函数的返回值是否为 true，来选择属性的值是否需要改变。

代码示例：

```
class PostHierarchy {
    var grade: String by Delegates.vetoable("T0", {
        property, oldValue, newValue ->
        true
    })

    var notChangeGrade: String by Delegates.vetoable("T0", {
        property, oldValue, newValue ->
        false
    })
}
```

测试代码：

```
ph.grade = "T1"
ph.grade = "T2"
ph.grade = "T3"
println(ph.grade) // T3

ph.notChangeGrade = "T1"
ph.notChangeGrade = "T2"
ph.notChangeGrade = "T3"
println(ph.notChangeGrade) // T0
```

我们可以看出，当 onChange 函数返回值是 false 的时候，对属性 notChangeGrade 的赋值都没有生效，依然是原来的默认值 T0。

5. 非空属性委托（Delegates. notNull）

我们也可以使用委托来实现属性的非空限制：

```
var name: String by Delegates.notNull()
```

这样 name 属性就被限制为不能为 null，如果被赋值 null，编译器直接报错：

```
ph.name = null // error
Null can not be a value of a non-null type String
```

6. 属性委托给 Map 映射

我们也可以把属性委托给 Map：

```
class Account(val map: Map<String, Any?>) {
    val name: String by map
    val password: String by map
}
```

测试代码：

```
val account = Account(mapOf(
            "name" to "admin",
            "password" to "admin"
    ))

println("Account(name=${account.name}, password = ${account.password})")
```

输出：

```
Account(name=admin, password = admin)
```

如果是可变属性，这里也可以把只读的 Map 换成 MutableMap：

```
class MutableAccount(val map: MutableMap<String, Any?>) {
    var name: String by map
    var password: String by map
}
```

测试代码：

```
val maccount = MutableAccount(mutableMapOf(
            "name" to "admin",
            "password" to "admin"
))

maccount.password = "root"
println("MutableAccount(name=${maccount.name}, password = ${maccount.
password})")
```

输出：

```
MutableAccount(name=admin, password = root)
```

7.14 本章小结

总的来说，Kotlin 相比于 Java 的面向对象编程，增加不少有趣的功能与特性支持。

例如有了数据类，我们就不再需要像 Java 中那样写那么多样板化的 getter，setter（虽然这些 IDE 都基本一键生成，但是现在我们不再需要生成了）。这使得我们代码写起来更加方便快捷了。例如，有了 object 对象类，我们可以直接写出单例模式的类。

另外，Kotlin 直接在语言层面支持委托，从而使得代理模式的编程更加自然优雅。

我们知道，在 Java 8 中，引进了对函数式编程的支持：Lambda 表达式、Function 接口、stream API 等，而在 Kotlin 中，对函数式编程的支持更加全面丰富，代码写起来也更加简洁优雅。下一章中，我们来一起学习 Kotlin 的函数式编程。

本章示例代码：https://github.com/EasyKotlin/chatper7_oop



第 8 章 Chapter 8

函数式编程

值就是函数，函数就是值。所有函数都消费函数，所有函数都生产函数。

函数作为 Kotlin 中的“一等公民”，可以像其他对象一样作为函数的输入与输出。关于对函数式编程的支持，相对于 Scala 的学院派风格，Kotlin 则是纯工程派：实用性、简洁性上都要比 Scala 好。

本章我们一起来学习函数式编程的简史、Lambda 演算、Y 组合子与递归等核心函数式的编程思想，以及在 Kotlin 中如何使用函数式风格编程。其中重点介绍了 Kotlin 中的高阶函数、Lambda 表达式、闭包等核心语法，并给出相应的实例说明。

8.1 函数式编程概述

“函数式编程”又称泛函编程，是一种“编程范式”（programming paradigm），也就是如何编写程序的方法论。它的基础是 λ 演算（lambda calculus）。λ 演算可以接受函数当作输入（参数）和输出（返回值）。

和指令式编程相比，函数式编程的思维方式更加注重函数的计算。它的主要思想是把问题的解决方案写成一系列嵌套的函数调用。

就像在 OOP 中，一切皆是对象，编程的是由对象交合创造的世界；而在 FP 中，一切皆是函数，编程的是由函数交合创造的世界。

函数式编程中最古老的例子莫过于 1958 年被创造出来的 Lisp 了。Lisp 由约翰·麦卡锡（John McCarthy，1927—2011）在 1958 年基于 λ 演算所创造，采用抽象数据列表与递归

作符号演算来衍生人工智能。较现代的例子包括 Haskell、ML、Erlang 等。现代的编程语言对函数式编程都做了不同程度的支持，例如 Kotlin、JavaScript、Coffee Script、PHP、Perl、Python、Ruby、C/C++/ C#、Java、Scala、Clojure、Lua 等（这将是一个不断增长的列表）。如果加入函数式语言要求的模式匹配、无副作用等要求，那么剩下的就是纯函数式语言，比较常见的有 Haskell 和 Clean 等。

函数式语言在 Java 虚拟机（JVM）平台上也迅速崭露头角，例如 Kotlin、Scala、Clojure；NET 平台也不例外，例如 F#。

函数式（FP）编程思想是一种历史悠久的思想（相对于整个计算机科学而言），其发展历史见图 8-1。

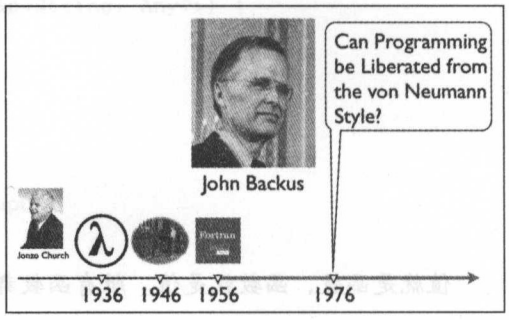


图 8-1 函数式编程时间轴

1920 年，Schönfinkel 提出了组合子逻辑（combinatory logic）。后来 Curry Brooks Haskell 一生都在研究组合子，并开创了这个研究领域。λ 演算中用单参数函数来表示多个参数函数的方法称为柯里化（Currying），虽然 Curry 多次指出这个其实是 Schönfinkel 已经提出来的，不过其他人都是因为他用了才知道，所以这名字就这么定下来了。有三门编程语言以他的名字命名，分别是 Curry、Brooks、Haskell。

1933 年，Church 提出一套以纯 λ 演算为基础的逻辑，以期对数学进行形式化描述。λ 演算和递归函数理论就是函数式编程的基础。

1936 年，Alan Turing 和 Alonzo Church 两人同在 1936 年独立给出了“确定性问题”的否定答案。

1935 ~ 1936 年，这个时间段里，我们有了三个有效计算模型：通用图灵机、通用递归函数、λ 可定义。Rosser 在 1939 年证明了这三个模型是等效的。

1953 ~ 1957 年，这期间 John Backus 开发出了 FORTRAN（FORmula TRANslating）。1977 年他的图灵奖演讲题为“ Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs ”。

1956 年，John McCarthy 在做人工智能研究时，开发出一个代数列表处理（algebraic list processing）语言 LISP。因为是在 IBM 704 上进行的，所以 LISP 的表处理函数才会有 car/cdr 这样的名字，其实是取 IBM 704 机器字的不同部分，即 c=content of, r=register number, a=address part, d=decrement part。

8.1.1 函数式编程与面向对象编程

1. OOP 中的设计模式与高阶函数

我们熟知的在 OOP 编程范式中的设计模式，其实在函数式编程中基本上都可以使用高

阶函数来实现。形象化的表达如图 8-2 所示。

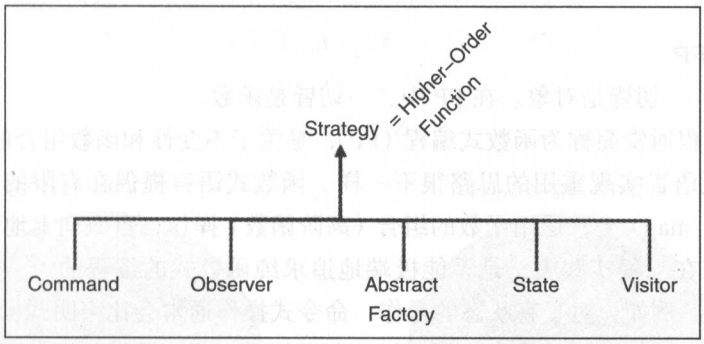


图 8-2　在函数式编程中用高阶函数来实现



提示　在面向对象的命令式（imperative）编程语言里面，构建整个世界的基础是类和类之间沟通用的消息，这些都可以用类图（class diagram）来表述。《设计模式：可复用面向对象软件的基础》（Design Patterns: Elements of Reusable Object-Oriented Software，作者是 ErichGamma、Richard Helm、Ralph Johnson、John Vlissides 四个人，简称 GoF）一书中，在每一个模式的说明里都附上了至少一幅类图。

OOP 的世界提倡开发者针对具体问题建立专门的数据结构，相关的专门操作行为以“方法”的形式附加在数据结构上，自顶向下地来构建其编程世界。

OOP 追求的是万事万物皆对象的理念，自然地弱化了函数。例如，函数无法像普通数据那样来传递（OOP 在函数指针上的约束），所以在 OOP 中有各种各样、五花八门的设计模式。

GoF 所著的《设计模式：可复用面向对象软件的基础》从面向对象设计的角度出发，通过对封装、继承、多态、组合等技术的反复使用，提炼出一些可重复使用的面向对象设计技巧，而多态在其中又是重中之重。

多态、面向接口编程、依赖反转等术语，描述的思想其实是相同的。这种反转模式实现了模块与模块之间的解耦。这样的架构是健壮的，而为了实现这样的健壮系统，在系统架构中基本都需要使用多态性。

绝大部分设计模式的实现都离不开多态性的思想。换一种说法就是，这些设计模式背后的本质其实就是 OOP 的多态性，而 OOP 中的多态本质上又是受约束的函数指针。

引用 Charlie Calverts 对多态的描述：“多态性是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。”

简单地说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。而我们在

OOP 中的那么多的设计模式，其实就是在 OOP 的多态性的约束规则下，对这些函数指针的调用模式的总结。

2. OOP 与 FP

在 OOP 中，一切皆是对象。在 FP 中，一切皆是函数。

面向函数编程通常简称为函数式编程（FP），是关于不变性和函数组合的一种编程范式。

函数式编程语言实现重用的思路很不一样。函数式语言提倡在有限的几种关键数据结构（如 list、set、map）上，运用函数的组合（高阶函数）操作，自底向上地来构建世界。

当然，我们在工程实践中，是不能极端地追求纯函数式的编程的。一个简单的原因就是：性能和效率。例如：对于有状态的操作，命令式操作通常会比声明式操作更有效率。纯函数式编程是解决某些问题的伟大工具，但是在另外的一些问题场景中并不适用，因为副作用总是真实存在。

我们在 7.1.3 节中已经知道了 OOP 的编程风格。而在 FP 中，是通过函数组合来构造其逻辑系统。FP 倾向于把软件分解为其需要执行的行为或操作，而且通常采用自底向上的方法。函数式编程也提供了非常强大的对事物进行抽象和组合的能力。

在 FP 里面，函数是“一等公民”（first-class）。它们可以像 1、2、“hello”、true、对象之类的“值”一样，在任意位置诞生，通过变量、参数和数据结构传递到其他地方，可以在任何位置被调用。

而在 OOP 中，很多所谓面向对象设计模式，都是因为面向对象语言没有 first-class function（对应的是多态性），所以导致了每个函数必须被包在一个对象里面（受约束的函数指针）才能传递到其他地方。

3. 匀称的数据结构 + 匀称的算法

在面向对象式的编程中，一切皆是对象（偏重数据结构、数据抽象，轻算法）。我们把它叫作胖数据结构 – 瘦算法（FDS-TA）。

在面向函数式的编程中，一切皆是函数（偏重算法，轻数据结构）。我们把它叫作瘦数据结构 – 胖算法（TDS-FA）。

可是，这个世界很复杂，怎么能说“一切皆是……”呢？真实的编程世界，自然是匀称的数据结构结合匀称的算法（SDS-SA）来创造的。

在编程中，我们不可能使用纯的对象（对象的行为方法其实就是函数）或者纯的函数（调用函数的对象、函数操作的数据其实就是数据结构）来创造一个完整的世界。如果数据结构是阴，算法是阳，那么在解决实际问题时，往往需要阴阳交合。还是那句经典的话：

程序 = 匀称的数据结构 + 匀称的算法

我们用一幅图（见图 8-3）来简单说明。

4. 函数与映射

一切皆是映射。函数式编程的代码主要就是“对映射的描述”。我们说组合是编程的本

质，其实，组合就是建立映射关系。

一个函数无非就是从输入到输出的映射，写成数学表达式就是：

$$f: X -> Y$$

$$p: Y -> Z$$

$$p(f): X -> Z$$

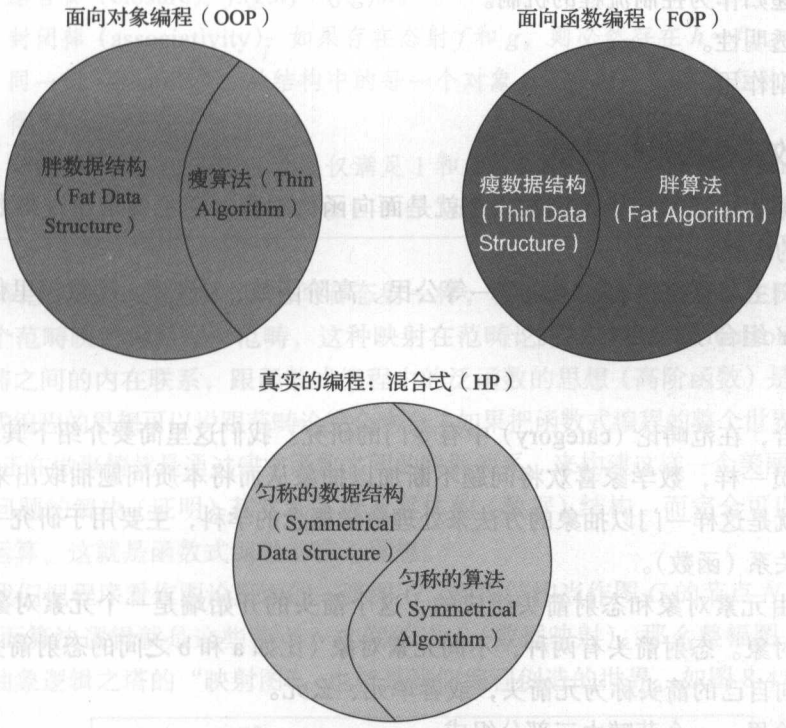


图 8-3　自然是匀称的数据结构结合匀称的算法

用编程语言表达就是：

```
fun f(x:X) : Y{}
fun p(y:Y) : Z{}
fun fp(f: (X)->Y, p: (Y)->Z) : Z {
    return {x -> p(f(x))}
}
```

8.1.2　函数式编程基本特性

在经常被引用的论文“Why Functional Programming Matters”中，作者 John Hughes 说明了模块化是成功编程的关键，而函数编程可以极大地改进模块化。

在函数编程中，我们有一个内置的框架来开发更小的、更简单的和更一般化的模块，然后将它们组合在一起。

函数编程的一些基本特点包括：

- ❑ 函数是“一等公民”。
- ❑ 闭包（closure）和高阶函数（higher order function）。
- ❑ Lambda 演算与函数柯里化（Currying）。
- ❑ 懒惰计算（lazy evaluation）。
- ❑ 使用递归作为控制流程的机制。
- ❑ 引用透明性。
- ❑ 没有副作用。

8.1.3 函数式编程核心思想

函数式编程中，一切皆是函数，也就是面向函数编程。它主要是通过函数的组合算子来实现程序的逻辑。

本节我们主要介绍范畴、函数是一等公民、高阶函数、λ 演算、函数柯里化、闭包、递归函数以及 Y 组合子等相关内容。

1. 范畴

关于组合，在范畴论（category）中有专门的研究。我们这里简要介绍下其思想。

与程序员一样，数学家喜欢将问题不断加以抽象从而将本质问题抽取出来加以论证解决，范畴论就是这样一门以抽象的方法来处理数学概念的学科，主要用于研究一些数学结构之间的映射关系（函数）。

范畴是由元素对象和态射箭头组成的，这个箭头的开始端是一个元素对象，目的地也是一个元素对象。态射箭头有两种，不同元素对象（比如 a 和 b 之间的态射箭头）称为组合箭头，而指向自己的箭头称为元箭头，或者单元、幺元。

在范畴论里，一个范畴由三部分组成：

- ❑ 对象（object）
- ❑ 态射（morphism）
- ❑ 组合（composition）操作符

如果我们把函数式编程看作一个范畴，那么函数就是其中的对象，函数之间的组合就是对应的态射与组合操作符。

范畴的对象 这里的对象可以看成是一类东西，数学上的群、环以及有理数、无理数等都可以归为一个对象。对应到编程语言里，可以理解为一个类型，比如整型、布尔型等。在函数式编程范畴里，范畴的对象就是函数。

范畴的态射 态射就是映射，用箭头表示。始端是一个元素对象，终端也是一个元素对象：

$$f\colon a \rightarrow b$$

范畴的组合 范畴中的组合操作符用点 (.) 表示，用于将态射进行组合。例如，如果有态射

$$f: A \to B,\ g: B \to C,$$

则

$$g.f: A \to C.$$



一个结构要想成为一个范畴，除了必须包含上述三点，它还要满足以下三个限制：

1）结合律（closure）：$f.(g.h) = (f.g).h$。

2）封闭律（associativity）：如果存在态射 f 和 g，则必然存在 $h = f.g$。

3）同一律（identity）：对结构中的每一个对象 a，存在一个单位态射 $I: A \to A$，使得 $I.a = a$。

在群论中，一个群 G 如果仅满足 1 和 2，则称 G 是一个半群（semigroup）；如果满足 1、2、3，则称 G 是一个含幺半群（monoid）。

在范畴论里正如对象与对象之间有态射一样，范畴与范畴之间也存在映射关系，从而可以将一个范畴映射为另一个范畴，这种映射在范畴论中叫作函子（functor）。函子反映了不同的范畴之间的内在联系，跟函数式编程中的泛函数的思想（高阶函数）是相同的。

函数式编程的思想可以说跟范畴论完全吻合。如果把函数式编程的整个世界看作一个对象，那么 FP 真正在做事情就是通过建立函数之间的映射关系，来构建这样一个美丽的编程世界。

很多问题的解决（证明）其实都不涉及具体的（数据）结构，而完全可以只依赖映射之间的组合运算。这就是函数式编程的核心思想。

如果我们把程序看作图论里面的一张图 G，数据结构当作图 G 的节点 N（数据结构、存储状态），而算法逻辑就是这些节点 N 之间的边 E（数据映射），那么整幅图 $G(N, E)$ 就是一幅美妙的抽象逻辑之塔的“映射图”，也就是我们编程创造的世界，如图 8-4 所示。

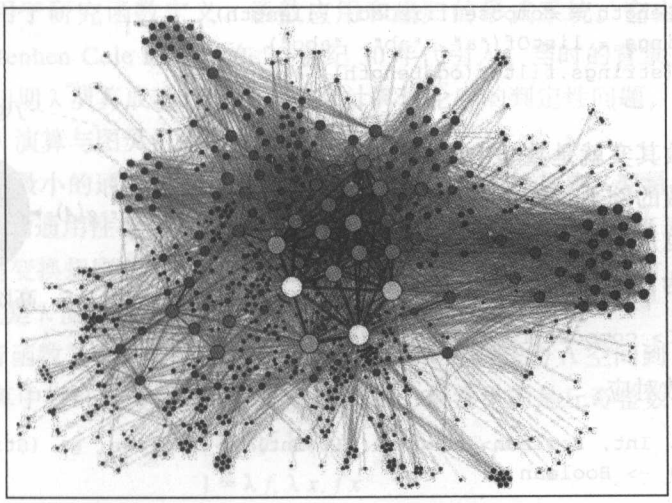


图 8-4　抽象逻辑之塔的“映射图”

2. 函数是“一等公民”

在函数式编程中，函数是“一等公民”。所谓“一等公民”，有时称为“闭包”或者“仿函数”对象，指的是函数与其他数据类型一样，处于平等地位，可以赋值给其他变量，也可以作为参数传入另一个函数，或者作为别的函数的返回值。这个以函数为参数的概念，跟 C 语言中的函数指针类似。

举例来说，下面代码中的 print 变量就是一个函数（没有函数名），可以作为另一个函数的参数：

```
>>> val print = fun(x:Any){println(x)}
>>> listOf(1,2,3).forEach(print)
1
2
3
```

3. 高阶函数

FP 语言支持高阶函数，高阶函数就是多阶映射。高阶函数用另一个函数作为其输入参数，也可以返回一个函数作为输出。

代码示例：

```
fun isOdd(x: Int) = x % 2 != 0
fun length(s: String) = s.length

fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

测试代码：

```
fun main(args: Array<String>) {
    val oddLength = compose(::isOdd, ::length)
    val strings = listOf("a", "ab", "abc")
    println(strings.filter(oddLength)) // [a, abc]
}
```

compose 函数其实就是数学中的复合函数的概念，这是一个高阶函数的例子：传入的两个参数 f 和 g 都是函数，其返回值也是函数。如图 8-5 所示。

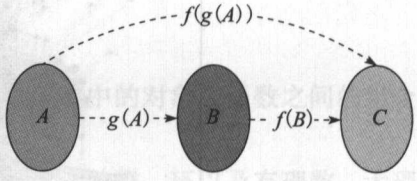


图 8-5 高阶函数的例子

这里的函数为：

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C
```

其中类型参数对应：

```
fun <String, Int, Boolean> compose(f: (Int) -> Boolean, g: (String) -> Int):
    (String) -> Boolean
```

这里的 (Int) -> Boolean、(String) -> Int、(String) -> Boolean 都是函数类型。

其实，从映射的角度看就是二阶映射。对 [a, ab, abc] 中每个元素 x 先映射成长度 $g(x)$ = 1, 2, 3，再进行第二次映射 $f(g(x))$ %2 != 0。判断长度是否为奇数，返回值是 true 的被过滤出来。

有了高阶函数，我们可以用优雅的方式进行模块化编程。

另外，高阶函数满足结合律，如图 8-6 所示。

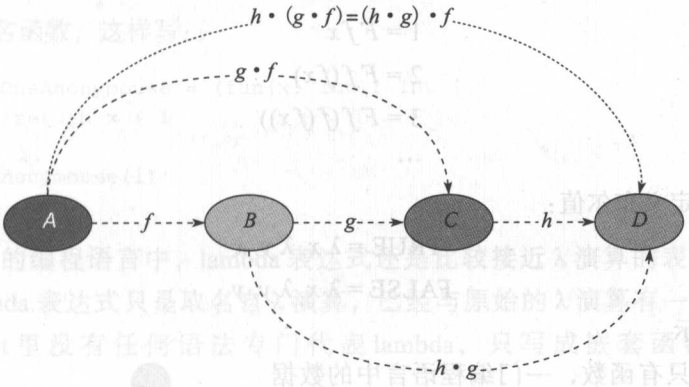


图 8-6　高阶函数的结合律

4. λ 演算

演算是函数式语言的基础。在 λ 演算的基础上发展起来的 π 演算、χ 演算，成为近年来并发程序的理论工具，许多经典的并发程序模型就是以 π 演算为框架的。λ 演算的神奇之处在于，通过最基本的函数抽象和函数应用法则，配套以适当的技巧，便能够构造出任意复杂的可计算函数。

λ 演算是一套用于研究函数定义、函数应用和递归的形式系统。它由 Alonzo Church（1903—1995）和 Stephen Cole Kleene 在 20 世纪 30 年代引入。当时的背景是解决函数可计算的本质性问题，初期 λ 演算成功地解决了在可计算理论中的判定性问题，后来根据丘奇 - 图灵论题，证明了 λ 演算与图灵机是等价的。

λ 演算可以称为最小的通用程序设计语言。它包括一条变换规则（变量替换）和一条函数定义方式，λ 演算的通用性在于，任何一个可计算函数都能用这种形式来表达和求值。

λ 演算强调的是变换规则的运用，这里的变换规则本质上就是函数映射。

Lambda 表达式是 λ 演算的一部分。

λ 演算中一切皆函数，全体 λ 表达式构成 Λ 空间，λ 表达式为 Λ 空间到 Λ 空间的函数。

例如，在 λ 演算中有许多方式都可以定义自然数，最常见的是丘奇整数，定义如下：

$$0 = \lambda f.\, \lambda x.\, x$$

$$1 = \lambda f.\, \lambda x.\, fx$$

$$2 = \lambda f.\, \lambda x.\, f(fx)$$

$$3 = \lambda f.\, \lambda x.\, f(f(f x))$$
$$\cdots$$

数学家们都崇尚简洁，只用一个关键字 λ 来表示对函数的抽象。

其中的 $\lambda f.\, \lambda x.$，λf 是抽象出来的函数，λx 是输入参数，点（.）语法用来分割参数表和函数体。为了更简洁，我们简记为 F，那么上面的丘奇整数定义简写为：

$$0 = F\, x$$
$$1 = F\, f\, x$$
$$2 = F\, f\, (f x)$$
$$3 = F\, f\, (f\, (f x))$$
$$\cdots$$

使用 λ 演算定义布尔值：

$$\text{TRUE} = \lambda x.\, \lambda y.\, x$$
$$\text{FALSE} = \lambda x.\, \lambda y.\, y$$

如图 8-7 所示。

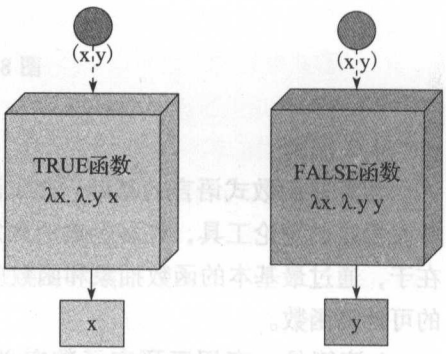


图 8-7 使用 λ 演算定义布尔值

在 λ 演算中只有函数，一门编程语言中的数据类型（比如 boolean、number、list 等）都可以使用纯 λ 演算来实现。我们不用关心数据的值是什么，重点是我们能对这个值做什么操作。

使用 λ 演算定义一个恒等函数 I：

$$I = \lambda x\, .\, x$$

使用 Kotlin 代码来写，如下：

```
>>> val I = {x:Int -> x}
>>> I(0)
0
>>> I(1)
1
>>> I(100)
100
```

对 I 而言任何一个 x 都是它的不动点（即对某个函数 $f(x)$ 存在这样的一个输入 x，使得函数的输出仍旧等于输入的 x。形式化的表示即为 $f(x) = x$）。

再例如，下面的 λ 表达式表示将 x 映射为 x+1：

$$\lambda x\, .\, x + 1$$

测试代码：

```
( λ x . x + 1) 5
```

将输出 6。

在 Kotlin 中，如果使用 Lambda 表达式，我们这样写：

```
>>> val addOneLambda = {
...         x: Int ->
...         x + 1
...     }
>>> addOneLambda(1)
2
```

如果使用匿名函数，这样写：

```
>>> val addOneAnonymouse = (fun(x: Int): Int {
...         return x + 1
...     })
>>> addOneAnonymouse(1)
2
```

在一些古老的编程语言中，lambda 表达式还是比较接近 λ 演算的表达式。在现代程序语言中的 lambda 表达式只是取名自 λ 演算，已经与原始的 λ 演算有一定的差别了。例如，在 JavaScript 里没有任何语法专门代表 lambda，只写成嵌套函数 function{return function{...}}。

5. 函数柯里化

很多基于 λ 演算的程序语言，比如 ML 和 Haskell，都习惯用柯里化（currying）的手法来表示函数。比如，如果在 Haskell 里面这样写一个函数：

$$f\,x\,y = x + y$$

然后你就可以这样把链表里的每个元素加上 2：

```
map (f 2) [1, 2, 3]
```

它会输出 [3, 4, 5]。

柯里化用一元函数来组合成多元函数。比如，上面的函数 f 的定义在 Scheme 里面相当于：

```
(define f (lambda (x) (lambda (y) (+ x y))))
```

它表示函数 f 接受一个参数 x，返回另一个函数（没有名字）。这个匿名函数如果再接受一个参数 y，就会返回 $x + y$。所以上面的例子里面，(f 2) 返回的是一个匿名函数，它会把 2 加到自己的参数上面返回。所以把它映射到 [1, 2, 3]，我们就得到了 [3, 4, 5]。

我们再使用 Kotlin 中的函数式编程来举例说明。首先，我们看下普通的二元函数的写法：

```
fun add(x: Int, y: Int): Int {
    return x + y
}

add(1, 2) // 输出 3
```

这种写法最简单，只有一层映射。

柯里化的写法：

```
fun curryAdd(x: Int): (Int) -> Int {
    return { y -> x + y }
}

curryAdd(1)(2)// 输出 3
```

我们先传入参数 $x = 1$，返回函数 curryAdd(1) = 1 + y；然后传入参数 $y = 2$，返回最终的值 curryAdd(1)(2) = 3。

当然，我们也有 λ 表达式的写法：

```
val lambdaCurryAdd = {
        x: Int ->
        {
            y: Int ->
            x + y
        }
    }

lambdaCurryAdd(1)(2)  // 输出 3
```

这个做法其实来源于最早的 λ 演算的设计。因为 λ 演算的函数都只有一个参数，所以为了表示多参数的函数，Haskell Curry（数学家和逻辑学家）发明了这个方法。

不过在编码实践中，柯里化的工程实用性和简洁性都不是那么好。大量使用柯里化，会导致代码可读性降低，复杂性增加，并且还可能因此引入意想不到的错误。所以在讲求工程实践性能的 Kotlin 语言中，古老而美丽的理论也许能够给我带来思想的启迪，但是未必那么理想。

6. 闭包

Ruby 之父松本行弘在《代码的未来》一书中对闭包（Closure）这样解释：闭包就是把函数以及变量包起来，使得变量的生存周期延长。

简单讲，闭包就是一个代码块，用 { } 包起来。此时，程序代码也就成了数据，可以被一个变量引用（与 C 语言的函数指针比较类似）。闭包的最典型应用是实现回调函数（callback）。

闭包包含以下两个组成部分：

- 要执行的代码块（由于自由变量被包含在代码块中，这些自由变量以及它们引用的对象没有被释放）。
- 自由变量的作用域。

在 PHP、Scala、Scheme、Common Lisp、Smalltalk、Groovy、JavaScript、Ruby、

Python、Go、Lua、objective c、swift 以及 Java（Java8 及以上）等语言中都能找到对闭包不同程度的支持。支持 Lambda 表达式的语言一般也会支持闭包。



虽然各个语言对 Lambda 表达式、匿名函数、闭包三者在概念理解和实现上会有些不同，但是总的思想是一致的：为了更简洁、更灵活的变量作用域，为了方便我们写代码而准备的“语法糖”。

7. 惰性计算

除了高阶函数、闭包、Lambda 表达式等概念，FP 还引入了惰性计算的概念。惰性计算（尽可能延迟表达式求值）是许多函数式编程语言的特性。

在惰性计算中，表达式不是在绑定到变量时立即计算，而是在求值程序需要产生表达式的值时进行计算。

一个惰性计算的例子是生成无穷 Fibonacci 列表的函数，对第 n 个 Fibonacci 数的计算相当于只是从可能的无穷列表中提取一项。

8. 递归函数

递归指的是一个函数在其定义中直接或间接调用自身的一种方法，它通常把一个大型的复杂的问题转化为一个与原问题相似的规模较小的问题来解决（复用函数自身），这样可以极大地减少代码量。

递归分为两个阶段：

1）**递推**：把复杂问题的求解推到比原问题简单一些的问题的求解。

2）**回归**：当获得最简单的情况后，逐步返回，依次得到复杂的解。

递归的能力在于用有限的语句来定义对象的无限集合。

使用递归要注意两点：

- 递归就是在过程或函数里面调用自身。
- 在使用递归时，必须有一个明确的递归结束条件，称为递归出口。

下面我们举例说明。

阶乘函数 fact(n) 一般这样递归地定义：

```
fact(n) = if n=0 then 1 else n * fact(n-1)
```

我们使用 Kotlin 代码实现这个函数如下：

```
fun factorial(n: Int): Int {
    println("factorial() called!  n=$n")
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

测试代码：

```
@Test
fun testFactorial() {
    Assert.assertTrue(factorial(0) == 1)
    Assert.assertTrue(factorial(1) == 1)
    Assert.assertTrue(factorial(3) == 6)
    Assert.assertTrue(factorial(10) == 3628800)
}
```

输出：

```
factorial() called!  n=0
factorial() called!  n=1
...
factorial() called!  n=1
factorial() called!  n=0
BUILD SUCCESSFUL in 24s
6 actionable tasks: 5 executed, 1 up-to-date
```

我们可以看到在 factorial 计算的过程中，函数不断地调用自身，然后不断地展开，直到最后到达了终止的 n==0。这是递归的原则之一，就是在递归的过程中，传递的参数一定要不断地接近终止条件，在上面的例子中就是 *n* 的值不断减少，直至最后为 0。

再举个 Fibonacci 数列的例子。

Fibonacci 数列用数学中的数列的递归表达式定义如下：

```
fibonacci (0) = 0
fibonacci (1) = 1
fibonacci (n) = fibonacci (n - 1) + fibonacci (n - 2)
```

我们使用 Kotlin 代码实现它：

```
fun fibonacci(n: Int): Int {
    if (n == 1 || n == 2) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

测试代码：

```
@Test
fun testFibonacci() {
    Assert.assertTrue(fibonacci(1) == 1)
    Assert.assertTrue(fibonacci(2) == 1)
    Assert.assertTrue(fibonacci(3) == 2)
    Assert.assertTrue(fibonacci(4) == 3)
    Assert.assertTrue(fibonacci(5) == 5)
    Assert.assertTrue(fibonacci(6) == 8)
}
```

外篇：Scheme 中的递归写法

因为 Scheme 程序中充满了一对对嵌套的小括号，这些嵌套的符号体现了最基本的数学思想——递归。所以，为了多维度地来理解递归，我们给出 Scheme 中的递归写法：

```
(define factorial
    (lambda (n)
        (if (= n 0)
            1
            (* n (factorial (- n 1))))))


(define fibonacci
    (lambda (n)
        (cond ((= n 0) 0)
              ((= n 1) 1)
              (else (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))))
```

其中关键字 lambda 表明我们定义的（即任何封闭的开括号立即离开 λ 及其相应的关闭括号）是一个函数。

Lambda 演算和函数式语言的计算模型天生较为接近，Lambda 表达式一般是这些语言必备的基本特性。

Scheme 是 Lisp 方言，遵循极简主义哲学，有着独特的魅力。Scheme 的一个主要特性是可以像操作数据一样操作函数调用。

9. Y 组合子

她让人绞尽脑汁，也琢磨不定！她让人心力交瘁，又百般回味！她，看似平淡，却深藏玄机！她，貌不惊人，却天下无敌！她是谁？她就是 Y 组合子（Y-Combinator），不动点组合子中最著名的一个。

在现代编程语言中，函数都是具名的，而在传统的 λ 演算中，函数都是没有名字的。这样就出现了一个问题——如何在 λ 演算中实现递归函数，即匿名递归函数。Haskell B. Curry（编程语言 Haskell 就是以此人命名的）发现了一种不动点组合子——Y 组合子，用于解决匿名递归函数实现的问题。



对于任意函数 g，可以通过推导得到 $Yg = g(Yg)$（（高阶）函数的不动点），从而证明 λ 演算是“图灵完备”的。Y 组合子的重要性由此可见一斑。

Y 组合子的定义是：

$$Y = \lambda f.(\lambda x.f(x\,x))\,(\lambda x.f(x\,x))$$

Y 组合子让我们可以定义匿名的递归函数。Y 组合子是 λ 演算的一部分，也是函数式编

程的理论基础。仅仅通过 Lambda 表达式这个最基本的原子实现循环迭代。Y 组合子本身是函数，其输入也是函数（在 Lisp 中连程序都是函数），颇有道生一、一生二、二生三、三生万物的韵味。

举个例子说明：我们先使用类 C 语言中较为熟悉的 JavaScript 来实现一个 Y 组合子函数，因为 JavaScript 语言的动态特性，使得该实现相比许多需要声明各种类型的语言要简洁许多：

```
function Y(f) {
    return (function (g) {
        return g(g);
    })(function (g) {
        return f(function (x) {
            return g(g)(x);
        });
    });
}

var fact = Y(function (rec) {
    return function (n) {
        return n == 0 ? 1 : n * rec(n - 1);
    };
});
```

我们使用了 Y 函数组合一段匿名函数代码，实现了一个匿名的递归阶乘函数。

直接将这两个函数放到浏览器的 Console 中去执行，我们将看到如下输出：

```
fact(10)
3628800
```

这个 Y 函数相当烧脑。要是在 Clojure(JVM 上的 Lisp 方言）中，这个 Y 函数实现如下：

```
(defn Y [r]
 ((fn [f] (f f))
 (fn [f]
 (r (fn [x] ((f f) x))))))
```

使用 Scheme 语言来表达：

```
(define Y
    (lambda (f)
        ((lambda (x) (f (lambda (y) ((x x) y))))
            (lambda (x) (f (lambda (y) ((x x) y)))))))
```

我们可以看出，使用 Scheme 语言表达的 Y 组合子跟原生的 λ 演算表达式基本一样。

用 CoffeeScript 实现的 Y 组合子如下：

```
coffee> Y = (f) -> ((x) -> (x x)) ((x) -> (f ((y) -> ((x x) y))))
[Function]
```

这个看起就相当简洁优雅了。我们使用这个 Y 组合子实现一个匿名递归的 Fibonacci 函数：

```
coffee> fib = Y (f) -> (n) ->  if n < 2 then n else f(n-1) + f(n-2)
[Function]
coffee> index = [0..10]
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
coffee> index.map(fib)
[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

实现一个匿名递归阶乘函数：

```
coffee> fact = Y (f)  ->(n) -> if n==0 then 1 else n*f(n-1)
[Function]
coffee> fact(10)
3628800
```

上面的 Coffee 代码的命令行 REPL 运行环境搭建非常简单：

```
$ npm install -g coffee-script
$ coffee
coffee>
```



对 CoffeeScript 感兴趣的读者，可以参考 http://coffee-script.org/。

但是，Y 组合子要是使用 OOP 语言编程范式，就要复杂许多。为了更加深刻地认识 OOP 与 FP 编程范式，我们使用 Java 8 以及 Kotlin 的实例来说明。这里使用 Java 给出示例的原因，是为了对比后面的章节将会介绍的 Kotlin 与 Java 的互操作。

首先我们使用 Java 的匿名内部类实现 Y 组合子：

```
package com.easy.kotlin;

/**
 * Created by jack on 2017/7/9.
 */
public class YCombinator {
    public static Lambda<Lambda> yCombinator(final Lambda<Lambda> f) {
        return new Lambda<Lambda>() {
            @Override
            public Lambda call(Object input) {
                final Lambda<Lambda> u = (Lambda<Lambda>)input;
                return u.call(u);
            }
        }.call(new Lambda<Lambda>() {
            @Override
            public Lambda call(Object input) {
                final Lambda<Lambda> x = (Lambda<Lambda>)input;
                return f.call(new Lambda<Object>() {
                    @Override
                    public Object call(Object input) {
                        return x.call(x).call(input);
```

```
                }
            });
        }
    });
}

public static void main(String[] args) {
    Lambda<Lambda> y = yCombinator(new Lambda<Lambda>() {
        @Override
        public Lambda call(Object input) {
            final Lambda<Integer> fab = (Lambda<Integer>)input;
            return new Lambda<Integer>() {
                @Override
                public Integer call(Object input) {
                    Integer n = Integer.parseInt(input.toString());
                    if (n < 2) {
                        return Integer.valueOf(1);
                    } else {
                        return n * fab.call(n - 1);
                    }
                }
            };
        }
    });
    System.out.println(y.call(10));//输出: 3628800
}

interface Lambda<E> {
    E call(Object input);
}
}
```

这里定义了一个 Lambda<E> 类型，然后通过 E call（Object input）方法实现自调用，方法实现里有多处转型以及嵌套调用。逻辑比较复杂，代码可读性也比较差。当然，这个问题本身也比较复杂。

我们使用 Java 8 的 Lambda 表达式来改写下匿名内部类：

```
package com.easy.kotlin;

/**
 * Created by jack on 2017/7/9.
 */
public class YCombinator2 {

    public static Lambda<Lambda> yCombinator2(final Lambda<Lambda> f) {
        return ((Lambda<Lambda>)(Object input) -> {
            final Lambda<Lambda> u = (Lambda<Lambda>)input;
            return u.call(u);
        }).call(
            ((Lambda<Lambda>)(Object input) -> {
```

```
                final Lambda<Lambda> v = (Lambda<Lambda>)input;
                return f.call((Lambda<Object>)(Object p) -> {
                    return v.call(v).call(p);
                });
            })
        );

    }
    public static void main(String[] args) {
        Lambda<Lambda> y2 = yCombinator2(
            (Lambda<Lambda>)(Object input) -> {
                Lambda<Integer> fab = (Lambda<Integer>)input;
                return (Lambda<Integer>)(Object p) -> {
                    Integer n = Integer.parseInt(p.toString());
                    if (n < 2) {
                        return Integer.valueOf(1);
                    } else {
                        return n * fab.call(n - 1);
                    }
                };
            });

        System.out.println(y2.call(10));//输出: 3628800
    }

    interface Lambda<E> {
        E call(Object input);
    }

}
```

最后，我们使用Kotlin的对象表达式（顺便复习一下上一章节的相关内容）实现Y组合子：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/7/9.
 *
 * lambda f. (lambda x. (f(x x)) lambda x. (f(x x)))
 *
 * OOP YCombinator
 *
 */


object YCombinatorKt {

    fun yCombinator(f: Lambda<Lambda<*>>): Lambda<Lambda<*>> {
```

```
        return object : Lambda<Lambda<*>> {
            override fun call(n: Any): Lambda<*> {
                val u = n as Lambda<Lambda<*>>
                return u.call(u)
            }
        }.call(object : Lambda<Lambda<*>> {

            override fun call(n: Any): Lambda<*> {
                val x = n as Lambda<Lambda<*>>

                return f.call(object : Lambda<Any> {
                    override fun call(n: Any): Any {
                        return x.call(x).call(n)!!
                    }
                })
            }

        }) as Lambda<Lambda<*>>
    }

    @JvmStatic fun main(args: Array<String>) {

        val y = yCombinator(object : Lambda<Lambda<*>> {

            override fun call(n: Any): Lambda<*> {
                val fab = n as Lambda<Int>

                return object : Lambda<Int> {

                    override fun call(n: Any): Int {
                        val n = Integer.parseInt(n.toString())
                        if (n < 2) {
                            return Integer.valueOf(1)
                        } else {
                            return n * fab.call(n - 1)
                        }
                    }
                }
            }
        })

        println(y.call(10)) //输出: 3628800
    }

    interface Lambda<E> {
        fun call(n: Any): E
    }
}
```

当然，对于函数式编程也完全支持的 Kotlin 也有 FP 风格的 Y 组合子实现：

```
/**
 * Created by jack on 2017/7/9.
 *
 * lambda f. (lambda x. (f(x x)) lambda x. (f(x x)))
 *
 * FP YCombinator
 */

//为了方便易懂，使用 X 用做函数 (X)->Int 的别名
typealias X = (Int) -> Int
//G 递归引用 G 自己
interface G : Function1<G, X>
//create a fun G from lazy blocking
fun G(block: (G) -> X) = object : G {
    //调用函数自身 `block(g)` like as `g(g)`
    override fun invoke(g: G) = block(g)
}

typealias F = Function1<X, X>
fun Y(f: F) = (fun(g: G) = g(g))(G { g -> f({ x -> g(g)(x) }) })

val fact = Y {
    rec ->
    {
        n ->
        if (n == 0) 1 else n * rec(n - 1)
    }
}
val fib = Y { f ->
    {
        n ->
        if (n == 1 || n == 2) 1 else f(n - 1) + f(n - 2)

    }
}


fun main(args: Array<String>) {
    println(fact(10))
    println(fib(10))

    for (i in 1..10) {
        println("$i!= ${fact(i)}")
    }

    for (i in 1..10) {
        println("fib($i) = ${fib(i)}")
    }
}
```

我们定义接口 G 继承了函数接口 Function1<G, X>：

```
interface G : Function1<G, X>
```

Function1 接口继承自 kotlin.Function 接口：

```
public interface Function<out R>
```

Function1 有一个抽象算子函数 invoke，用来调用入参 p1：

```
public interface Function1<in P1, out R> : kotlin.Function<R> {
    public abstract operator fun invoke(p1: P1): R
}
```

我们定义 G 函数，入参是 block 函数 (G) -> X，block 函数的入参又是 G 类型，通过 invoke 函数来实现调用自身：

```
fun G(block: (G) -> X) = object : G {
    //调用函数自身 `block(g)` like as `g(g)`
    override fun invoke(g: G) = block(g)
}
```

这样，我们就可以实现一个 Y 组合子函数了：

```
typealias F = Function1<X, X>

fun Y(f: F) = (fun(g: G) = g(g))(G { g -> f({ x -> g(g)(x) }) })
```

我们通过 Y 组合子就可以定义匿名阶乘递归函数和匿名 Fibonacci 数列递归函数了。代码示例如下：

```
val fact = Y {
    rec ->
    {
        n ->
        if (n == 0) 1 else n * rec(n - 1)
    }
}
val fib = Y { f ->
    {
        n ->
        if (n == 1 || n == 2) 1 else f(n - 1) + f(n - 2)

    }
}
```

测试代码：

```
fun main(args: Array<String>) {
    val square: X = { x -> x * x }
    println(square(9))

    println(fact(10))
```

```
    println(fib(10))

    for (i in 1..10) {
        println("$i!= ${fact(i)}")
    }

    for (i in 1..10) {
        println("fib($i) = ${fib(i)}")
    }
}
```



关于 Y 组合子的更多实现，可以参考 https://gist.github.com/Jason-Chen-2017/88e13b63fa5b7c612fddf999739964b0。另外，关于 Y 组合子的原理介绍，推荐看《The Little Schemer》这本书。

从上面的例子，我们可以看出 OOP 中的接口以及多态类型，跟 FP 中的函数所表达的本质上是一个东西，这个东西到底是什么呢？我们姑且称之为“编程之道”吧！

Y 组合子给我们提供了一种方法，让我们在一个只支持 first-class 函数但是没有内建递归的编程语言里完成递归。所以 Y 组合子给我们展示了一个语言完全可以定义递归函数，即使这个语言的定义一点也没提到递归。它给我们展示了一件美妙的事：函数式编程可以让我们做到我们从来不认为可以做到的事。

严谨而精巧的 λ 演算体系，从最基本的概念“函数”入手，创造出一个绚烂而宏伟的世界，这不能不说是人类思维的骄傲。

10. 没有“副作用”：纯函数

严格的无副作用的编程几乎只出现在纯函数式程序中，现实中的 OOP 编程场景难免触及副作用。因此在这里所说的避免副作用主要指的是：

- 函数本身的运行稳定可预期。
- 函数的运行不对外部环境造成意料外的污染。

纯函数的输入输出数据流全是显式的。显式的意思是，函数与外界交换数据只有一个唯一渠道：参数和返回值。意思是说，函数从函数外部接受的所有输入信息都通过参数传递到该函数内部，函数输出到函数外部的所有信息都通过返回值传递到该函数外部。

对于无副作用的纯函数而言，输入同样的参数，执行后总能得到同样的结果，这种幂等性使得一个函数无论在什么上下文中运行、运行多少次，最后的结果总是可预期的。这让用户非常放心，不用关心函数逻辑的细节，不用考虑是否应该在某个特定的时机调用，也不用记录调用的次数，等等。

纯函数的好处主要有以下几点：

- 无状态，线程安全，不需要线程同步。
- 纯函数相互调用组装起来的函数还是纯函数。

❑ 运行时可以对纯函数的运算结果进行缓存，加快运算速度。

如果一个函数通过隐式方式从外界获取数据，或者向外部输出数据，那么，该函数就不是纯函数，而叫作非纯函数，非纯函数是有副作用的。隐式的意思是，函数通过参数和返回值以外的渠道和外界进行数据交换。比如，读取全局变量、修改全局变量都叫作以隐式的方式和外界进行数据交换；比如，利用 I/O API（输入输出系统函数库）读取配置文件，或者输出到文件，打印到屏幕，都叫做隐式的方式和外界进行数据交换。



当然，实际应用中不做 I/O 是不可能的。因此，编程过程中，函数式编程只要求把 I/O 限制到最小，不要有不必要的读写行为，保持计算过程的单纯性。

11. 引用透明性

在函数式编程中，我们常常还会提到一个词：引用透明。引用透明的概念与纯函数很接近：如果一个表达式对于相同的输入总是有相同的结果，并且不修改程序其他部分的状态，那么这个表达式是引用透明的。

由前面纯函数的定义可以看到，由于函数调用也是表达式的一种，因此任何纯函数的调用都满足引用透明。

8.2 在 Kotlin 中使用函数式编程

前文中我们在函数式编程的世界里遨游了一番，现在我们把思绪收回来，放在 Kotlin 中的函数式编程中来。

严格的面向对象的观点，使得很多问题的解决方案变得较为笨拙。为了将一行有用的代码包装到 Runnable 或者 Callable 这两个 Java 中最流行的函数式示例中，我们不得不去写五六行模板范例代码。为了让事情简单化（在 Java 8 中，增加 Lambda 表达式的支持），我们在 Kotlin 中使用普通的函数来替代函数式接口。事实上，函数式编程中的函数比 C 语言中的函数或者 Java 中的方法都要强大得多。

Kotlin 支持函数作为一等公民，支持高阶函数、Lambda 表达式等。我们不仅可以把函数当作普通变量一样传递、返回，还可以把它分配给变量、放进数据结构或者进行一般性的操作。函数可以是未经命名的，也就是匿名的。我们也可以直接把一段代码丢到 {} 中，这就是闭包。

在前面的章节中，其实我们已经涉及一些关于函数的知识，我们将在这里系统地学习一下 Kotlin 的函数式编程。

8.2.1 Kotlin 中的函数

首先，我们来看下 Kotlin 中函数的概念。

1. 函数声明

Kotlin 中的函数使用 fun 关键字声明：

```
fun double(x: Int): Int {
    return 2*x
}
```

如果一个函数 triple 需要传入一个 double 函数，返回一个从 Int 映射到 Int 的函数，那么这个这个函数的类型声明如下：

```
fun triple (double: (Int)->Int): (Int)->Int  {
    // 函数体
    return {x -> double(x) + x}
}
```

这里的 (Int)->Int 是函数类型声明，表示一个从 Int 映射到 Int 的函数。

使用的时候，我们需要先声明一个函数字面量：

```
val double = fun(x:Int):Int = x+x
```

然后这样调用这个 triple 函数：

```
>>> triple(double)(3)
9
>>> triple(double)(4)
12
>>> triple(double)(10)
30
```

2. 函数用法

调用函数使用传统的方法：

```
fun test() {
        val doubleTwo = double(2)
        println("double(2) = $doubleTwo")
    }
```

输出：

```
double(2) = 4
```

调用成员函数使用点表示法：

```
object FPBasics {

    fun double(x: Int): Int {
        return 2 * x
    }

    fun test() {
```

```
        val doubleTwo = double(2)
        println("double(2) = $doubleTwo")
    }
}

fun main(args: Array<String>) {
    FPBasics.test()
}
```

我们这里直接用 object 对象 FPBasics 来演示。

8.2.2 扩展函数

通过“扩展”声明完成一个类的新功能扩展，而无需继承该类或使用设计模式（例如，装饰者模式）。一个扩展 String 类的 swap 函数的例子如下：

```
fun String.swap(index1: Int, index2: Int): String {
    val charArray = this.toCharArray()
    val tmp = charArray[index1]
    charArray[index1] = charArray[index2]
    charArray[index2] = tmp

    return charArrayToString(charArray)
}

fun charArrayToString(charArray: CharArray): String {
    var result = ""
    charArray.forEach { it -> result = result + it }
    return result
}
```

这个 this 关键字在扩展函数内部对应到接收者对象（传过来的在点符号前的对象）。现在，我们对任意 String 调用该函数：

```
val str = "abcd"
val swapStr = str.swap(0, str.lastIndex)
println("str.swap(0, str.lastIndex) = $swapStr")
```

输出：

```
str.swap(0, str.lastIndex) = dbca
```

8.2.3 中缀函数

在以下场景中，函数还可以用中缀表示法调用：

- 成员函数或扩展函数。
- 只有一个参数。
- 用 infix 关键字标注。

例如，给 Int 定义扩展：

```
infix fun Int.shl(x: Int): Int {
 ...
}
```

用中缀表示法调用扩展函数：1 shl 2 等同于 1.shl(2)。

8.2.4　函数参数

函数参数使用 Pascal 表示法定义，即 name: type。参数用逗号隔开。每个参数必须显式指定其类型：

```
fun powerOf(number: Int, exponent: Int): Int {
    return Math.pow(number.toDouble(), exponent.toDouble()).toInt()
}
```

测试代码：

```
val eight = powerOf(2, 3)
println("powerOf(2,3) = $eight")
```

输出：

```
powerOf(2,3) = 8
```

1. 默认参数

函数参数可以有默认值，当省略相应的参数时使用默认值。这可以减少重载数量：

```
fun add(x: Int = 0, y: Int = 0): Int {
    return x + y
}
```

默认值通过类型后面的 = 及给出的值来定义。

测试代码：

```
val zero = add()
val one = add(1)
val two = add(1, 1)
println("add() = $zero")
println("add(1) = $one")
println("add(1, 1) = $two")
```

输出：

```
add() = 0
add(1) = 1
add(1, 1) = 2
```

另外，覆盖带默认参数的函数时，总是使用与基类型方法相同的默认参数值。

当覆盖一个带有默认参数值的方法时，签名中不带默认参数值：

```
open class DefaultParamBase {
    open fun add(x: Int = 0, y: Int = 0): Int {
        return x + y
    }
}

class DefaultParam : DefaultParamBase() {
    override fun add(x: Int, y: Int): Int { //不能有默认值
        return super.add(x, y)
    }
}
```

2. 命名参数

可以在调用函数时使用命名的函数参数。当一个函数有大量的参数或默认参数时这会非常方便。

给定以下函数：

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = false,
             wordSeparator: Char = ' ') {

}
```

我们可以使用默认参数来调用它：

```
reformat(str)
```

然而，当使用非默认参数调用它时，该调用看起来就像：

```
reformat(str, true, true, false, '_')
```

使用命名参数我们可以使代码更具有可读性：

```
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
```

并且如果我们不需要所有的参数：

```
reformat(str, wordSeparator = '_')
```

3. 可变数量的参数

函数的参数（通常是最后一个）可以用 vararg 修饰符标记：

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) //ts is an Array
        result.add(t)
    return result
}
```

允许将可变数量的参数传递给函数：

```
val list = asList(1, 2, 3)
```

8.2.5　函数返回类型

具有块代码体的函数必须始终显式指定返回类型，除非它们旨在返回 Unit。

Kotlin 不推断具有块代码体的函数的返回类型，因为这样的函数在代码体中可能有复杂的控制流，并且返回类型对于读者（有时对于编译器）也是不明显的。

如果一个函数不返回任何有用的值，它的返回类型是 Unit。Unit 是一种只有一个 Unit 值的类型。这个值不需要显式返回：

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    //`return Unit` 或者 `return` 是可选的
}
```

Unit 返回类型声明也是可选的。上面的代码等同于：

```
fun printHello(name: String?) {
    .....
}
```

8.2.6　单表达式函数

当函数返回单个表达式时，可以省略花括号并且在 = 符号之后指定代码体即可：

```
fun double(x: Int): Int = x * 2
```

当返回值类型可由编译器推断时，显式声明返回类型是可选的：

```
fun double(x: Int) = x * 2
```

8.2.7　函数作用域

在 Kotlin 中函数可以在文件顶层声明，这意味着不需要像一些语言（如 Java、C# 或 Scala）那样创建一个类来保存一个函数。此外除了顶层函数，Kotlin 中的函数也可以声明为局部函数、成员函数以及扩展函数。

局部函数（嵌套函数）指一个函数在另一个函数内部：

```
fun sum(x: Int, y: Int, z: Int): Int {
    val delta = 0;
    fun add(a: Int, b: Int): Int {
        return a + b + delta
    }
    return add(x + add(y, z))
}
```

局部函数可以访问外部函数（即闭包）中的局部变量 delta：

```
println("sum(1,2,3) = ${sum(0, 1, 2, 3)}")
```

输出：

```
sum(1,2,3) = 6
```

成员函数是在类或对象内部定义的函数：

```
class Sample() {
    fun foo() { print("Foo") }
}
```

成员函数以点表示法调用：

```
Sample().foo() //创建类 Sample 实例并调用 foo
```

8.2.8 泛型函数

函数可以有泛型参数，通过在函数名前使用尖括号指定。

例如 Iterable 的 map 函数如下所示：

```
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    return mapTo(ArrayList<R>(collectionSizeOrDefault(10)), transform)
}
```

8.2.9 高阶函数

高阶函数是将函数用作参数或返回值的函数。例如，Iterable 的 filter 函数：

```
public inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T> {
    return filterTo(ArrayList<T>(), predicate)
}
```

它的输入参数 predicate: (T) -> Boolean 就是一个函数。其中，函数类型声明的语法是：

```
(X)->Y
```

表示这个函数是从类型 X 到类型 Y 的映射。即这个函数输入 X 类型，输出 Y 类型。

这个函数我们这样调用：

```
fun isOdd(x: Int): Boolean {
    return x % 2 == 1
}

val list = listOf(1, 2, 3, 4, 5)
list.filter(::isOdd)
```

其中，:: 用来引用一个函数。

8.2.10　匿名函数

我们也可以使用匿名函数来实现这个 predicate 函数：

```
list.filter((fun(x: Int): Boolean {
    return x % 2 == 1
}))
```

8.2.11　Lambda 表达式

我们也可以直接使用更简单的 Lambda 表达式来实现一个 predicate 函数：

```
list.filter {
    it % 2 == 1
}
```

- lambda 表达式总是被大括号 {} 括着。
- 其参数（如果有的话）在 -> 之前声明（参数类型可以省略）。
- 函数体（如果存在的话）在 -> 后面。

如果 lambda 是 filter 函数的唯一参数（入参 lambda 表达式也是一个函数），则调用中的圆括号可以省略，也就是说上面的写法与如下代码等价：

```
list.filter({
    it % 2 == 1
})
```

使用 Lambda 表达式定义一个函数字面值如下所示：

```
>>> val sum = { x: Int, y: Int -> x + y }
>>> sum(1,1)
2
```

我们再使用嵌套的 Lambda 表达式来定义一个柯里化的 sum 函数：

```
>>> val sum = {x:Int ->  {y:Int -> x+y }}
>>> sum
(kotlin.Int) -> (kotlin.Int) -> kotlin.Int
>>> sum(1)(1)
2
```

8.2.12 it：单个参数的隐式名称

Kotlin 中另一个有用的约定是，如果函数字面值只有一个参数，那么它的声明可以省略（连同 ->），其名称是 it。

代码示例：

```
>>> val list = listOf(1,2,3,4,5)
>>> list.map { it * 2 }
[2, 4, 6, 8, 10]
```

8.2.13 闭包

Lambda 表达式、匿名函数以及局部函数和对象表达式都可以访问其“闭包”在外部作用域中声明的变量。我们也可以修改闭包中捕获的变量：

```
fun sumGTZero(c: Iterable<Int>): Int {
    var sum = 0
    c.filter { it > 0 }.forEach {
        sum += it
    }
    return sum
}

val list = listOf(1, 2, 3, 4, 5)
sumGTZero(list) //输出 15
```

我们再使用闭包来写一个使用 Java 中的 Thread 接口的例子：

```
fun closureDemo() {
    Thread({
        for (i in 1..10) {
            println("I = $i")
            Thread.sleep(1000)
        }

    }).start()

    Thread({
        for (j in 10..20) {
            println("J = $j")
            Thread.sleep(2000)
        }
        Thread.sleep(1000)
    }).start()
}
```

一个输出：

```
I = 1
J = 10
```

```
I = 2
I = 3
...
J = 20
```

8.2.14　带接收者的函数字面值

Kotlin 提供了使用指定的“接收者对象”调用函数字面值的功能。使用匿名函数的语法，我们可以直接指定函数字面值的接收者类型。

下面我们使用带接收者的函数类型声明一个变量，并在之后使用它。代码示例：

```
>>> val sum = fun Int.(other: Int): Int = this + other
>>> 1.sum(1)
2
```

当接收者类型可以从上下文推断时，lambda 表达式可以用作带接收者的函数字面值：

```
class HTML {
    fun body() {
        println("HTML BODY")
    }
}

fun html(init: HTML.() -> Unit): HTML { //HTML.() 中的 HTML 是接受者类型
    val html = HTML()  //创建接收者对象
    html.init()        //将该接收者对象传给该 lambda
    return html
}
```

测试代码：

```
html {
    body()
}
```

输出：

```
HTML BODY
```

使用这个特性，我们可以构建一个 HTML 的 DSL 语言。

8.2.15　具体化的类型参数

有时候我们需要访问一个参数类型：

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p.parent
    }
```

```
    @Suppress("UNCHECKED_CAST")
    return p as T?
}
```

在这里我们向上遍历一棵树并且检查每个节点是不是特定的类型。

这都没有问题，但是调用处不是很优雅：

```
treeNode.findParentOfType(MyTreeNode::class.java)
```

我们真正想要的只是传一个类型给该函数，即像这样调用它：

```
treeNode.findParentOfType<MyTreeNode>()
```

为能够这么做，内联函数支持“具体化的类型参数”，于是我们可以这样写：

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

我们使用 reified 修饰符来限定类型参数，现在可以在函数内部访问它了，几乎就像是一个普通的类一样。由于函数是内联的，不需要反射，正常的操作符如 !is 和 as 现在都能用了。

虽然在许多情况下可能不需要反射，但我们仍然可以对一个具体化的类型参数使用它：

```
inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
    println(membersOf<StringBuilder>().joinToString("\n"))
}
```

普通的函数（未标记为内联函数的）没有具体化参数。

8.2.16 尾递归 tailrec

Kotlin 支持一种称为尾递归的函数式编程风格。这允许一些通常用循环写的算法改用递归函数来写，而无堆栈溢出的风险。当一个函数用 tailrec 修饰符标记并满足所需的形式时，编译器会优化该递归，生成一个快速而高效的基于循环的版本：

```
tailrec fun findFixPoint(x: Double = 1.0): Double
        = if (x == Math.cos(x)) x else findFixPoint(Math.cos(x))
        // 函数必须将其自身调用作为它执行的最后一个操作
```

这段代码计算余弦的不动点，这是一个数学常数。它只是重复地从 1.0 开始调用 Math.cos，直到结果不再改变，产生 0.7390851332151607 的结果。最终代码相当于如下更传统风

格的代码：

```
private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (x == y) return y
        x = y
    }
}
```

要符合 tailrec 修饰符的条件，函数必须将其自身调用作为它执行的最后一个操作。在递归调用后有更多代码时，不能使用尾递归，并且不能用在 try/catch/finally 块中。在 JVM 后端中支持尾递归。

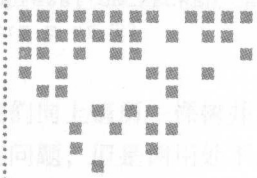
Kotlin 还为集合类引入了许多扩展函数。例如，使用 map() 和 filter() 函数可以流畅地操纵数据，具体的函数使用以及示例我们已经在第 5 章“集合类”中介绍过。

8.3　本章小结

函数作为 Kotlin 中的一等公民可以像其他对象一样作为函数的输入与输出，这也是 Java 程序员转到 Kotlin 时感觉变化最大、最难理解的一点。Kotlin 中的函数式编程和 Java 中最大的不同是函数形参，函数体可以用等号赋值给函数定义，函数和变量是同等的地位。而 Kotlin 对高阶函数的支持也相当简单优雅。我们使用高阶函数和 Lambda 表达式（闭包），可以更加优雅地进行模块化编程。

我们将在下一章中介绍 Kotlin 的轻量级线程——协程的相关知识，我们将看到，在 Kotlin 中，程序的逻辑可以在协程中顺序地表达，而底层库会为我们解决其异步性。

本章示例代码：https://github.com/EasyKotlin/chapter8_fp



Chapter 9 第 9 章

轻量级线程：协程

在常用的并发模型中，多进程、多线程、分布式是最普遍的，不过近些年来逐渐有一些语言以 first-class 或者 library 的形式支持基于协程的并发模型，例如 Scheme、Lua、Python、Perl、Go 等语言就以 first-class 的方式提供对协程的支持。

同样，Kotlin 也支持协程。本章我们首先简单介绍协程的基本概念，然后通过一些基础示例来学习有关协程的创建、执行、取消等操作。在本章的后半部分，我们主要探讨挂起函数的组合执行、协程上下文与调度器、通道与管道等相关内容。最后，对协程与线程进行了简单比较，简要介绍了 Kotlin 的协程 API 库。

9.1 协程的基本概念

从硬件发展来看，从最初的单核单 CPU 到单核多 CPU、多核多 CPU，似乎已经到了极限了，但是单核 CPU 性能却还在不断提升。

如果将程序分为 IO 密集型应用和 CPU 密集型应用，二者的发展历程大致如下：

- IO 密集型应用：多进程→多线程→事件驱动→协程。
- CPU 密集型应用：多进程→多线程。

如果说多进程对应多 CPU、多线程对应多核 CPU，那么事件驱动和协程则是在充分挖掘不断提高性能的单核 CPU 的潜力。

我们知道，常见的有性能瓶颈的 API，例如网络 IO、文件 IO、CPU 或 GPU 密集型任务等，要求调用者阻塞（blocking）直到它们完成才能进行下一步。后来，我们又使用异步回调的方式来实现非阻塞，但是异步回调代码写起来并不简单。

协程（coroutine）提供了一种避免阻塞线程并用更简单、更可控的操作替代线程阻塞的方法：协程挂起。

协程主要是让原来要使用“异步 + 回调方式”写出来的复杂代码，简化成可以用看似同步的方式（对线程的操作进一步抽象）。这样我们就可以按串行的思维模型去组织原本分散在不同上下文中的代码逻辑，而不需要去处理复杂的状态同步问题。

协程可以用来解决很多问题，比如嵌套回调、并发编程模型的实现等。

协程最早的描述是由 Melvin Conway 于 1958 年给出的：与主程序行为类似的子例程。此后他又在博士论文中给出了如下定义：

- 数据在后续调用中始终保持协程的局部。
- 当控制流程离开时，协程的执行被挂起，此后控制流程再次进入这个协程时，这个协程只应从上次离开挂起的地方继续。

协程的实现要维护一组局部状态，在重新进入协程前，保证这些状态不被改变，从而能顺利定位到之前的位置。

协程通过将复杂性放入库来简化异步编程。程序的逻辑可以在协程中顺序地表达，而底层库会为我们解决其异步性。该库可以将用户代码的相关部分包装为回调、订阅相关事件、在不同线程（甚至不同机器）上调度执行，而代码则保持如同顺序执行一样简单。

协程可以被挂起而无需阻塞线程。而线程阻塞的代价通常是昂贵的，尤其在高负载时，阻塞其中一个会导致一些重要的任务被延迟。另外，协程挂起几乎是无代价的，不需要上下文切换或者 OS 的任何其他干预。

最重要的是，协程的挂起可以在很大程度上由用户来控制，这样我们就可以在挂起时做一些我们想做的事情，例如记录日志、拦截处理等。

下面我们通过具体的代码实践来学习协程的相关内容。

9.2　协程的基本操作

Kotlin 协程是一种用户态的轻量级线程，它由协程构建器（launch coroutine builder）启动。下面我们通过示例介绍协程的基本操作。

9.2.1　搭建项目

首先，我们来新建一个支持编写协程代码的 Kotlin Gradle 项目。

使用 IDEA 创建完标准 Kotlin + Gradle 项目后，在配置文件 build.gradle 中，配置 kotlinx-coroutines-core 依赖：

添加 dependencies：

```
compile 'org.jetbrains.kotlinx:kotlinx-coroutines-core:0.16'
```

kotlinx-coroutines 还提供了下面的模块：

```
compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-jdk8',
    version: '0.16'
compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-nio',
    version: '0.16'
compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-reactive',
    version: '0.16'
```

使用 Kotlin 最新的 1.1.3-2 版本：

```
buildscript {
    ext.kotlin_version = '1.1.3-2'
    ...
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

其中，kotlin-gradle-plugin 是 Kotlin 集成 Gradle 的插件。

另外，配置一下 JCenter 的仓库：

```
repositories {
    jcenter()
}
```

9.2.2 简单协程示例

下面我们先来看一个简单的协程示例。

运行下面的代码：

```
fun firstCoroutineDemo0() {
    launch(CommonPool) {
        delay(3000L, TimeUnit.MILLISECONDS)
        println("Hello,")
    }
    println("World!")
    Thread.sleep(5000L)
}
```

你将会发现输出：

```
World!
Hello,
```

上面的这段代码：

```
launch(CommonPool) {
            delay(3000L, TimeUnit.MILLISECONDS)
            println("Hello,")
}
```

等价于：

```
launch(CommonPool, CoroutineStart.DEFAULT, {
```

```
        delay(3000L, TimeUnit.MILLISECONDS)
        println("Hello, ")
})
```

9.2.3　使用 launch 函数启动协程

launch 函数定义在 kotlinx.coroutines.experimental 下面：

```
public fun launch(
    context: CoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
    val newContext = newCoroutineContext(context)
    val coroutine = if (start.isLazy)
        LazyStandaloneCoroutine(newContext, block) else
        StandaloneCoroutine(newContext, active = true)
    coroutine.initParentJob(context[Job])
    start(block, coroutine, coroutine)
    return coroutine
}
```

launch 函数有 3 个入参：context、start、block，这些参数分别说明如下：

参数	说　明
context	协程上下文。
start	协程启动选项。
block	协程真正要执行的代码块，必须是 suspend 修饰的挂起函数。

这个 launch 函数返回一个 Job 类型，Job 是协程创建的后台任务，它持有该协程的引用。Job 接口实际上继承自 CoroutineContext 类型。一个 Job 有三种状态，见表 9-1。

表 9-1　Job 的三种状态

Job 状态	isActive	isCompleted
New（optional initial state）新建（可选的初始状态）	false	false
Active（default initial state）活动中（默认初始状态）	true	false
Completed（final state）已结束（最终状态）	false	true

也就是说，launch 函数以非阻塞（non-blocking）当前线程的方式，启动一个新的协程后台任务，并返回一个 Job 类型的对象作为当前协程的引用。

另外，这里的 delay() 函数类似于 Thread.sleep() 的功能，但更好的是：它不会阻塞线程，而只是挂起协程本身。当协程在等待时，线程将返回到池中，当等待完成时，协同将在池中的空闲线程上恢复。

9.2.4　CommonPool 对象

我们下面再来看一下 launch 函数的参数：CommonPool。

首先，CommonPool 是代表共享线程池，它的主要作用是来调度计算密集型任务的协程的执行。它的实现使用的是 java.util.concurrent 包下面的 API。它首先尝试创建一个 java.util.concurrent.ForkJoinPool（ForkJoinPool 是一个可以执行 ForkJoinTask 的 ExcuteService，采用了 work-stealing 模式：所有在池中的线程尝试去执行其他线程创建的子任务，这样很少有线程处于空闲状态，更加高效）；如果不可用，就使用 java.util.concurrent.Executors 来创建一个普通的线程池：Executors.newFixedThreadPool。相关代码在 kotlinx/coroutines/experimental/CommonPool.kt 中：

```
private fun createPool(): ExecutorService {
    val fjpClass = Try { Class.forName("java.util.concurrent.ForkJoinPool") }
        ?: return createPlainPool()
    if (!usePrivatePool) {
        Try { fjpClass.getMethod("commonPool")?.invoke(null) as? ExecutorService }
            ?.let { return it }
    }
    Try { fjpClass.getConstructor(Int::class.java).newInstance(defaultParallelism())
        as? ExecutorService }
        ?. let { return it }
    return createPlainPool()
}

private fun createPlainPool(): ExecutorService {
    val threadId = AtomicInteger()
    return Executors.newFixedThreadPool(defaultParallelism()) {
        Thread(it, "CommonPool-worker-${threadId.incrementAndGet()}").apply {
            isDaemon = true }
    }
}
```

CommonPool 对象类是 CoroutineContext 的子类型。CommonPool 的类型继承层次结构如图 9-1 所示。

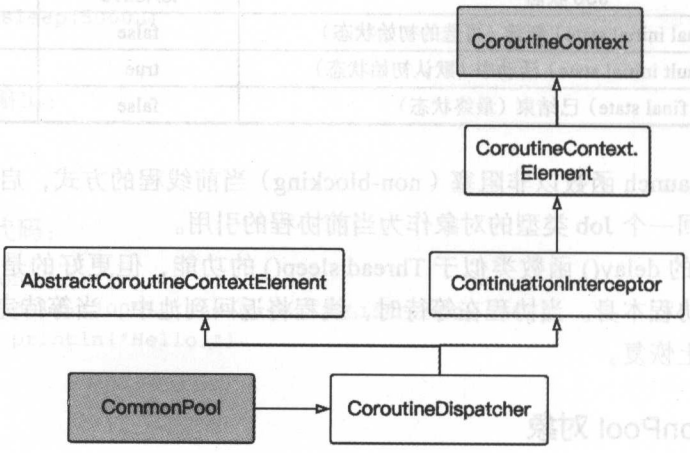


图 9-1 共享线程池类型层次结构

9.2.5　挂起函数

代码块中的 delay（3000L，TimeUnit.MILLISECONDS）函数，是一个用 suspend 关键字修饰的函数，我们称之为挂起函数。挂起函数只能从协程代码内部调用，普通的非协程代码不能调用。例如我们在协程代码中调用一个挂起函数，代码示例如下：

```
suspend fun runCoroutineDemo() {
    run(CommonPool) {
        delay(3000L, TimeUnit.MILLISECONDS)
        println("suspend,")
    }
    println("runCoroutineDemo!")
    Thread.sleep(5000L)
}

fun callSuspendFun() {
    launch(CommonPool) {
        runCoroutineDemo()
    }
}
```

如果我们用 Java 中的 Thread 类来写类似功能的代码，上面的代码可以写成这样：

```
fun threadDemo0() {
    Thread({
        Thread.sleep(3000L)
        println("Hello,")
    }).start()

    println("World!")
    Thread.sleep(5000L)
}
```

输出结果也是：

```
World!
Hello,
```

另外，我们不能使用 Thread 来启动协程代码。例如下面的写法编译器会报错：

```
/**
 * 错误反例：用线程调用协程 error
 */
fun threadCoroutineDemo() {
    Thread({
        delay(3000L, TimeUnit.MILLISECONDS) // error, Suspend functions are only
            allowed to be called from a coroutine or another suspend function
        println("Hello,")
    })
    println("World!")
```

```
    Thread.sleep(5000L)
}
```

9.2.6 runBlocking 函数

上面的例子中，我们给出的是使用非阻塞的 delay 函数，同时使用了阻塞的 Thread.sleep 函数，这样代码写在一起可读性不太好。让我们来使用纯的 Kotlin 协程代码来实现上面的例子，既有阻塞代码，又有非阻塞代码（不用 Thread）。

Kotlin 中提供了 runBlocking 函数来实现类似主协程的功能：

```
fun main(args: Array<String>) = runBlocking<Unit> {
    // 主协程
    println("${format(Date())}: T0")

    // 启动主协程
    launch(CommonPool) {
        // 在 common thread pool 中创建协程
        println("${format(Date())}: T1")
        delay(3000L)
        println("${format(Date())}: T2 Hello,")
    }
    println("${format(Date())}: T3 World!") // 当子协程被 delay，主协程仍然继续运行

    delay(5000L)

    println("${format(Date())}: T4")
}
```

运行结果：

```
14:37:59.640: T0
14:37:59.721: T1
14:37:59.721: T3 World!
14:38:02.763: T2 Hello,
14:38:04.738: T4
```

可以发现，运行结果与之前的示例是一样的，但是我们没有使用 Thread.sleep，我们只使用了非阻塞的 delay 函数。如果 main 函数不加 = runBlocking<Unit>，那么我们是不能在 main 函数体内调用 delay（5000L）的。

如果这个阻塞的线程被中断，runBlocking 抛出 InterruptedException 异常。

runBlocking 函数不是用来当做普通协程函数使用的，它的设计目的主要是用来桥接普通阻塞代码和挂起风格的（suspending style）的非阻塞代码，例如用在 main 函数中，或者用于测试用例代码中，代码示例：

```
@RunWith(JUnit4::class)
class RunBlockingTest {

    @Test fun testRunBlocking() = runBlocking<Unit> {
```

```
        //这样我们就可以在这里调用任何 suspend fun 了
        launch(CommonPool) {
            delay(3000L)
        }
        delay(5000L)
    }
}
```

9.2.7　等待一个协程任务执行完毕

我们先来看一段代码：

```
fun firstCoroutineDemo() {
    launch(CommonPool) {
        delay(3000L, TimeUnit.MILLISECONDS)
        println("[firstCoroutineDemo] Hello, 1")
    }

    launch(CommonPool, CoroutineStart.DEFAULT, {
        delay(3000L, TimeUnit.MILLISECONDS)
        println("[firstCoroutineDemo] Hello, 2")
    })
    println("[firstCoroutineDemo] World!")
}
```

运行这段代码，我们会发现只输出：

```
[firstCoroutineDemo] World!
```

这是为什么？

为了弄清上面的代码执行的内部过程，我们打印一些日志看下：

```
fun testJoinCoroutine() = runBlocking<Unit> {
    // Start a coroutine
    val c1 = launch(CommonPool) {
        println("C1 Thread: ${Thread.currentThread()}")
        println("C1 Start")
        delay(3000L)
        println("C1 World! 1")
    }

    val c2 = launch(CommonPool) {
        println("C2 Thread: ${Thread.currentThread()}")
        println("C2 Start")
        delay(5000L)
        println("C2 World! 2")
    }

    println("Main Thread: ${Thread.currentThread()}")
    println("Hello,")
    println("Hi,")
```

```
    println("c1 is active: ${c1.isActive}  ${c1.isCompleted}")
    println("c2 is active: ${c2.isActive}  ${c2.isCompleted}")

}
```

再次运行：

```
C1 Thread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
C1 Start
C2 Thread: Thread[ForkJoinPool.commonPool-worker-2,5,main]
C2 Start
Main Thread: Thread[main,5,main]
Hello,
Hi,
c1 is active: true  false
c2 is active: true  false
```

我们可以看到，这里的C1、C2代码也开始执行了，使用的是ForkJoinPool.commonPool-worker线程池中的worker线程。但是，我们在代码执行到最后打印出这两个协程的状态isCompleted都是false，这表明我们的C1、C2的代码在Main Thread结束的时刻（此时的运行main函数的Java进程也退出了），还没有执行完毕，然后就跟着主线程一起退出结束了。

所以我们可以得出结论：运行main()函数的主线程，必须要等到我们的协程完成之前结束，否则我们的程序在打印Hello, 1和Hello, 2之前就直接结束掉了。

我们怎样让这两个协程参与到主线程的时间顺序里呢？可以使用join，让主线程一直等到当前协程执行完毕再结束，例如下面的这段代码：

```
fun testJoinCoroutine() = runBlocking<Unit> {
    // Start a coroutine
    val c1 = launch(CommonPool) {
        println("C1 Thread: ${Thread.currentThread()}")
        println("C1 Start")
        delay(3000L)
        println("C1 World! 1")
    }

    val c2 = launch(CommonPool) {
        println("C2 Thread: ${Thread.currentThread()}")
        println("C2 Start")
        delay(5000L)
        println("C2 World! 2")
    }

    println("Main Thread: ${Thread.currentThread()}")
    println("Hello,")

    println("c1 is active: ${c1.isActive}  isCompleted: ${c1.isCompleted}")
    println("c2 is active: ${c2.isActive}  isCompleted: ${c2.isCompleted}")
```

```
    c1.join() //the main thread will wait until child coroutine completes
    println("Hi,")
    println("c1 is active: ${c1.isActive}  isCompleted: ${c1.isCompleted}")
    println("c2 is active: ${c2.isActive}  isCompleted: ${c2.isCompleted}")
    c2.join() //the main thread will wait until child coroutine completes
    println("c1 is active: ${c1.isActive}  isCompleted: ${c1.isCompleted}")
    println("c2 is active: ${c2.isActive}  isCompleted: ${c2.isCompleted}")
}
```

将会输出：

```
C1 Thread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
C1 Start
C2 Thread: Thread[ForkJoinPool.commonPool-worker-2,5,main]
C2 Start
Main Thread: Thread[main,5,main]
Hello,
c1 is active: true  isCompleted: false
c2 is active: true  isCompleted: false
C1 World! 1
Hi,
c1 is active: false  isCompleted: true
c2 is active: true  isCompleted: false
C2 World! 2
c1 is active: false  isCompleted: true
c2 is active: false  isCompleted: true
```

通常，出于良好的代码风格，我们会把一个单独的逻辑放到一个独立的函数中，我们可以重构上面的代码如下：

```
fun testJoinCoroutine2() = runBlocking<Unit> {
    //Start a coroutine
    val c1 = launch(CommonPool) {
        fc1()
    }

    val c2 = launch(CommonPool) {
        fc2()
    }
    ...
}

private suspend fun fc2() {
    println("C2 Thread: ${Thread.currentThread()}")
    println("C2 Start")
    delay(5000L)
    println("C2 World! 2")
}

private suspend fun fc1() {
    println("C1 Thread: ${Thread.currentThread()}")
```

```
    println("C1 Start")
    delay(3000L)
    println("C1 World! 1")
}
```

可以看出，这里的 fc1、fc2 函数是 suspend fun。

9.2.8 取消协程

我们知道，启动函数 launch 返回一个 Job 引用当前协程，该 Job 引用可用于取消正在运行协程：

```
fun testCancellation() = runBlocking<Unit> {
    val job = launch(CommonPool) {
        repeat(1000) { i ->
            println("I'm sleeping $i ... CurrentThread: ${Thread.currentThread()}")
            delay(500L)
        }
    }
    delay(1300L)
    println("CurrentThread: ${Thread.currentThread()}")
    println("Job is alive: ${job.isActive}  Job is completed: ${job.isCompleted}")
    val b1 = job.cancel() //cancels the job
    println("job cancel: $b1")
    delay(1300L)
    println("Job is alive: ${job.isActive}  Job is completed: ${job.isCompleted}")

    val b2 = job.cancel() //cancels the job, job already canceld, return false
    println("job cancel: $b2")

    println("main: Now I can quit.")
}
```

运行上面的代码，将会输出：

```
I'm sleeping 0 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 1 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 2 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
CurrentThread: Thread[main,5,main]
Job is alive: true  Job is completed: false
job cancel: true
Job is alive: false  Job is completed: true
job cancel: false
main: Now I can quit.
```

我们可以看出，当 job 还在运行时，isAlive 是 true，isCompleted 是 false。当调用 job.cancel 取消该协程任务，cancel 函数本身返回 true，此时协程的打印动作就停止了。此时，job 的状态是 isAlive 是 false，isCompleted 是 true。如果再次调用 job.cancel 函数，我们将会看到 cancel 函数返回的是 false。

9.2.9　协程取消失效

协程的所有 suspend 函数都是可以取消的。我们可以通过 job 的 isActive 状态来判断协程的状态，或者检查是否抛出 CancellationException 时取消。

例如，协程正在循环计算中，并且不检查协程当前的状态，那么调用 cancel 来取消协程将无法停止协程的运行，如下面的代码所示：

```
fun testCooperativeCancellation1() = runBlocking<Unit> {
    val job = launch(CommonPool) {
        var nextPrintTime = 0L
        var i = 0
        while (i < 20) { //computation loop
            val currentTime = System.currentTimeMillis()
            if (currentTime >= nextPrintTime) {
                println("I'm sleeping ${i++} ... CurrentThread: ${Thread.
                    currentThread()}")
                nextPrintTime = currentTime + 500L
            }
        }
    }
    delay(3000L)
    println("CurrentThread: ${Thread.currentThread()}")
    println("Before cancel, Job is alive: ${job.isActive}  Job is completed:
        ${job.isCompleted}")

    val b1 = job.cancel() //cancels the job
    println("job cancel1: $b1")
    println("After Cancel, Job is alive: ${job.isActive}  Job is completed:
        ${job.isCompleted}")

    delay(30000L)

    val b2 = job.cancel() //cancels the job, job already canceld, return false
    println("job cancel2: $b2")

    println("main: Now I can quit.")
}
```

运行上面的代码，输出：

```
I'm sleeping 0 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 1 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
...
I'm sleeping 6 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
CurrentThread: Thread[main,5,main]
Before cancel, Job is alive: true  Job is completed: false
job cancel1: true
After Cancel, Job is alive: false  Job is completed: true
I'm sleeping 7 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
```

```
...
I'm sleeping 18 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 19 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
job cancel2: false
main: Now I can quit.
```

我们可以看出，即使调用了 cancel 函数，当前的 job 状态 isAlive 是 false 了，但是协程的代码依然一直在运行，并没有停止。

9.2.10 计算代码协程的取消方式

有两种方法可以使计算代码取消成功。

方法一：显式检查取消状态 isActive

我们直接给出实现的代码：

```
fun testCooperativeCancellation2() = runBlocking<Unit> {
    val job = launch(CommonPool) {
        var nextPrintTime = 0L
        var i = 0
        while (i < 20) { // computation loop

            if (!isActive) {
                return@launch
            }

            val currentTime = System.currentTimeMillis()
            if (currentTime >= nextPrintTime) {
                println("I'm sleeping ${i++} ... CurrentThread: ${Thread.
                    currentThread()}")
                nextPrintTime = currentTime + 500L
            }
        }
    }
    delay(3000L)
    println("CurrentThread: ${Thread.currentThread()}")
    println("Before cancel, Job is alive: ${job.isActive}  Job is completed:
        ${job.isCompleted}")
    val b1 = job.cancel() // cancels the job
    println("job cancel1: $b1")
    println("After Cancel, Job is alive: ${job.isActive}  Job is completed:
        ${job.isCompleted}")

    delay(3000L)
    val b2 = job.cancel() // cancels the job, job already canceld, return false
    println("job cancel2: $b2")

    println("main: Now I can quit.")
}
```

运行这段代码，输出：

```
I'm sleeping 0 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 1 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 2 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 3 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 4 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 5 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 6 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
CurrentThread: Thread[main,5,main]
Before cancel, Job is alive: true  Job is completed: false
job cancel1: true
After Cancel, Job is alive: false  Job is completed: true
job cancel2: false
main: Now I can quit.
```

可见，现在这个循环可以被取消了。这里的 isActive 属性是 CoroutineScope 中的属性。这个接口的定义是：

```
public interface CoroutineScope {
    public val isActive: Boolean
    public val context: CoroutineContext
}
```

该接口用于通用协程构建器的接收器，以便协程中的代码可以方便地访问其 isActive 状态值（取消状态），以及其上下文 CoroutineContext 信息。

方法二：循环调用一个挂起函数 yield()

该方法实质上是通过 job 的 isCompleted 状态值来捕获 CancellationException 完成取消功能。

我们只需在 while 循环体中循环调用 yield() 来检查该 job 的取消状态，如果已经被取消，那么 isCompleted 值将会是 true，yield 函数就直接抛出 CancellationException 异常，从而完成取消的功能：

```
val job = launch(CommonPool) {
    var nextPrintTime = 0L
    var i = 0
    while (i < 20) { // computation loop

        yield()

        val currentTime = System.currentTimeMillis()
        if (currentTime >= nextPrintTime) {
            println("I'm sleeping ${i++} ... CurrentThread: ${Thread.currentThread()}")
            nextPrintTime = currentTime + 500L
        }
    }
}
```

运行上面的代码，输出：

```
I'm sleeping 0 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
I'm sleeping 1 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-2,5,main]
I'm sleeping 2 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-2,5,main]
I'm sleeping 3 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-3,5,main]
I'm sleeping 4 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-3,5,main]
I'm sleeping 5 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-3,5,main]
I'm sleeping 6 ... CurrentThread: Thread[ForkJoinPool.commonPool-worker-2,5,main]
CurrentThread: Thread[main,5,main]
Before cancel, Job is alive: true  Job is completed: false
job cancel1: true
After Cancel, Job is alive: false  Job is completed: true
job cancel2: false
main: Now I can quit.
```

如果我们想看看 yield 函数抛出的异常，我们可以加上 try catch 打印出日志：

```
try {
    yield()
} catch (e: Exception) {
    println("$i ${e.message}")
}
```

可以看到类似于 Job was cancelled 这样的信息。

这个 yield 函数的实现是：

```
suspend fun yield(): Unit = suspendCoroutineOrReturn sc@ { cont ->
    val context = cont.context
    val job = context[Job]
    if (job != null && job.isCompleted) throw job.getCompletionException()
    if (cont !is DispatchedContinuation<Unit>) return@sc Unit
    if (!cont.dispatcher.isDispatchNeeded(context)) return@sc Unit
    cont.dispatchYield(job, Unit)
    COROUTINE_SUSPENDED
}
```

如果调用此挂起函数时，当前协程的 Job 已经完成 (isActive = false, isCompleted = true)，当前协程将以 CancellationException 取消。

9.2.11 在 finally 中的协程代码

当我们取消一个协程任务时，如果有 try {...} finally {...} 代码块，那么 finally {…} 中的代码会正常执行完毕：

```
fun finallyCancelDemo() = runBlocking {
    val job = launch(CommonPool) {
        try {
            repeat(1000) { i ->
                println("I'm sleeping $i ...")
```

```
                delay(500L)
            }
        } finally {
            println("I'm running finally")
        }
    }
    delay(2000L)
    println("Before cancel, Job is alive: ${job.isActive}  Job is completed:
        ${job.isCompleted}")
    job.cancel()
    println("After cancel, Job is alive: ${job.isActive}  Job is completed:
        ${job.isCompleted}")
    delay(2000L)
    println("main: Now I can quit.")
}
```

运行这段代码，输出：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
I'm sleeping 3 ...
Before cancel, Job is alive: true  Job is completed: false
I'm running finally
After cancel, Job is alive: false  Job is completed: true
main: Now I can quit.
```

可以看出，在调用 cancel 之后，就算当前协程任务 Job 已经结束了，finally{...} 中的代码依然正常执行。

但是，如果我们在 finally{...} 中放入挂起函数：

```
fun finallyCancelDemo() = runBlocking {
    val job = launch(CommonPool) {
        try {
            repeat(1000) { i ->
                println("I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            println("I'm running finally")
            delay(1000L)
            println("And I've delayed for 1 sec ?")
        }
    }
    delay(2000L)
    println("Before cancel, Job is alive: ${job.isActive}  Job is completed:
        ${job.isCompleted}")
    job.cancel()
    println("After cancel, Job is alive: ${job.isActive}  Job is completed:
        ${job.isCompleted}")
```

```
    delay(2000L)
    println("main: Now I can quit.")
}
```

运行上述代码，我们将会发现只输出了一句：I'm running finally。因为主线程在挂起函数 delay（1000L）以及后面的打印逻辑还没执行完，就已经结束退出：

```
} finally {
    println("I'm running finally")
    delay(1000L)
    println("And I've delayed for 1 sec ?")
}
```

9.2.12 协程中不被取消的代码块

如果我们想要上面的例子中的 finally{...} 完整执行，不被取消函数操作所影响，我们可以使用 run 函数和 NonCancellable 上下文将相应的代码包装在 run (NonCancellable) {…} 中，如下面的示例所示：

```
fun testNonCancellable() = runBlocking {
    val job = launch(CommonPool) {
        try {
            repeat(1000) { i ->
                println("I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            run(NonCancellable) {
                println("I'm running finally")
                delay(1000L)
                println("And I've just delayed for 1 sec because I'm non-cancellable")
            }
        }
    }
    delay(2000L)
    println("main: I'm tired of waiting!")
    job.cancel()
    delay(2000L)
    println("main: Now I can quit.")
}
```

运行输出：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
I'm sleeping 3 ...
main: I'm tired of waiting!
I'm running finally
And I've just delayed for 1 sec because I'm non-cancellable
```

```
main: Now I can quit.
```

9.2.13　设置协程超时时间

我们通常取消协同执行的原因给协程的执行时间设定一个执行时间上限。我们也可以使用 withTimeout 函数来给一个协程任务的执行设定最大执行时间，超出这个时间，就直接终止掉。代码示例如下：

```
fun testTimeouts() = runBlocking {
    withTimeout(3000L) {
        repeat(100) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
}
```

运行上述代码，我们将会看到如下输出：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
I'm sleeping 3 ...
I'm sleeping 4 ...
I'm sleeping 5 ...
Exception in thread "main" kotlinx.coroutines.experimental.TimeoutException:
    Timed out waiting for 3000 MILLISECONDS
    at kotlinx.coroutines.experimental.TimeoutExceptionCoroutine.run(Scheduled.
        kt:110)
    at kotlinx.coroutines.experimental.EventLoopImpl$DelayedRunnableTask.
        invoke(EventLoop.kt:199)
    at kotlinx.coroutines.experimental.EventLoopImpl$DelayedRunnableTask.
        invoke(EventLoop.kt:195)
    at kotlinx.coroutines.experimental.EventLoopImpl.processNextEvent(EventLoop.
        kt:111)
    at kotlinx.coroutines.experimental.BlockingCoroutine.joinBlocking(Builders.
        kt:205)
    at kotlinx.coroutines.experimental.BuildersKt.runBlocking(Builders.kt:150)
    at kotlinx.coroutines.experimental.BuildersKt.runBlocking$default(Builders.
        kt:142)
    at com.easy.kotlin.CancellingCoroutineDemo.testTimeouts
        (CancellingCoroutineDemo.kt:169)
    at com.easy.kotlin.CancellingCoroutineDemoKt.main(CancellingCoroutineDemo.
        kt:193)
```

由 withTimeout 抛出的 TimeoutException 是 CancellationException 的一个子类。这个 TimeoutException 类型定义如下：

```
private class TimeoutException(
    time: Long,
```

```
    unit: TimeUnit,
    @JvmField val coroutine: Job
) : CancellationException("Timed out waiting for $time $unit")
```

如果需要在超时时执行一些附加操作，则可以把逻辑放在 try {…} catch (e: CancellationException) {…} 代码块中。例如：

```
try {
    ccd.testTimeouts()
} catch (e: CancellationException) {
    println("I am timed out!")
}
```

9.3 挂起函数

本节我们介绍挂起函数组合使用的各种方法，包括按默认顺序执行、使用异步并发执行。

9.3.1 按默认顺序执行

假设我们有两个在别处定义的挂起函数：

```
suspend fun doJob1(): Int {
    println("Doing Job1 ...")
    delay(1000L) //此处模拟我们的工作代码
    println("Job1 Done")
    return 10
}

suspend fun doJob2(): Int {
    println("Doing Job2 ...")
    delay(1000L) //此处模拟我们的工作代码
    println("Job2 Done")
    return 20
}
```

如果需要依次调用它们，只需要使用正常的顺序调用，因为协同中的代码（就像在常规代码中一样）是默认的顺序执行。下面的示例通过测量执行两个挂起函数所需的总时间来演示：

```
fun testSequential() = runBlocking<Unit> {
    val time = measureTimeMillis {
        val one = doJob1()
        val two = doJob2()
        println("[testSequential] 最终结果: ${one + two}")
    }
    println("[testSequential] Completed in $time ms")
}
```

执行上面的代码，我们将得到输出：

```
Doing Job1 ...
Job1 Done
Doing Job2 ...
Job2 Done
[testSequential] 最终结果: 30
[testSequential] Completed in 6023 ms
```

可以看出，我们的代码与普通的代码一样顺序执行下去了。

9.3.2　使用异步并发执行

上面的例子中，如果在调用 doJob1 和 doJob2 之间没有时序上的依赖关系，并且我们希望通过同时并发地执行这两个函数来更快地得到答案，那该怎么办呢？这个时候，我们就可以使用 async 来实现异步执行。代码示例如下：

```
fun testAsync() = runBlocking<Unit> {
    val time = measureTimeMillis {
        val one = async(CommonPool) { doJob1() }
        val two = async(CommonPool) { doJob2() }
        println("最终结果: ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
}
```

如果与上面同步的代码一起执行对比，我们可以看到如下输出：

```
Doing Job1 ...
Job1 Done
Doing Job2 ...
Job2 Done
[testSequential] 最终结果: 30
[testSequential] Completed in 6023 ms
Doing Job1 ...
Doing Job2 ...
Job1 Done
Job2 Done
[testAsync] 最终结果: 30
[testAsync] Completed in 3032 ms
```

可以看出，使用 async 函数，我们的两个 Job 并发地执行了，并发所用时间要比顺序执行快，速度要快将近两倍。因为，我们有两个任务在并发地执行。

从概念上讲，async 跟 launch 类似，它启动一个协程，与其他协程并发地执行。不同之处在于，launch 返回一个任务 Job 对象，不带任何结果值；而 async 返回一个延迟任务对象 Deferred，一种轻量级的非阻塞性 future，它表示后面会提供结果。

在上面的示例代码中，我们使用 Deferred 调用 await() 函数来获得其最终结果。另外，延迟任务 Deferred 也是 Job 类型，它继承自 Job，所以也有 isActive、isCompleted 属性，也

有 join()、cancel() 函数，因此我们也可以在需要时取消它。Deferred 接口定义如下：

```
public interface Deferred<out T> : Job {
    val isCompletedExceptionally: Boolean
    val isCancelled: Boolean
    public suspend fun await(): T
    public fun <R> registerSelectAwait(select: SelectInstance<R>, block:
        suspend (T) -> R)
    public fun getCompleted(): T
    @Deprecated(message = "Use `isActive`", replaceWith = ReplaceWith("isActive"))
    public val isComputing: Boolean get() = isActive
}
```

其中，常用的属性和函数说明如下：

名　称	说　明
isCompletedExceptionally	当协程在计算过程中有异常 failed 或被取消，返回 true。这也意味着 isActive 等于 false，同时 isCompleted 等于 true。
isCancelled	如果当前延迟任务被取消，返回 true。
suspend fun await()	等待此延迟任务完成，而不阻塞线程；如果延迟任务完成，则返回结果值或引发相应的异常。

延迟任务对象 Deferred 的状态与对应的属性值如表 9-2 所示。

表 9-2　延迟任务对象 Deferred 的状态与对应的属性值

状　态	isActive	isCompleted	isCompletedExceptionally	isCancelled
New（可选初始状态）	false	false	false	false
Active（默认初始状态）	true	false	false	false
Resolved（最终状态）	false	true	false	false
Failed（最终状态）	false	true	true	false
Cancelled（最终状态）	false	true	true	true

9.4　协程上下文与调度器

到这里，我们已经看到了下面这些启动协程的方式：

```
launch(CommonPool) {...}
async(CommonPool) {...}
run(NonCancellable) {...}
```

这里的 CommonPool 和 NonCancellable 是协程上下文（coroutine contexts）。本小节简单介绍一下自定义协程上下文。

9.4.1　调度和线程

协程上下文包括一个协程调度程序，它可以指定由哪个线程来执行协程。调度器可以

将协程的执行调度到一个线程池，限制在特定的线程中；也可以不做任何限制，让它无约束地运行。请看下面的示例：

```
fun testDispatchersAndThreads() = runBlocking {
    val jobs = arrayListOf<Job>()
    jobs += launch(Unconfined) {
        //未作限制 -- 将会在 main thread 中执行
        println("Unconfined: I'm working in thread ${Thread.currentThread()}")
    }
    jobs += launch(context) {
        //父协程的上下文: runBlocking coroutine
        println("context: I'm working in thread ${Thread.currentThread()}")
    }
    jobs += launch(CommonPool) {
        //调度指派给 ForkJoinPool.commonPool
        println("CommonPool: I'm working in thread ${Thread.currentThread()}")
    }
    jobs += launch(newSingleThreadContext("MyOwnThread")) {
        //将会在这个协程自己的新线程中执行
        println("newSingleThreadContext: I'm working in thread ${Thread.
            currentThread()}")
    }
    jobs.forEach { it.join() }
}
```

运行上面的代码，我们将得到以下输出（可能按不同的顺序）：

```
Unconfined: I'm working in thread Thread[main,5,main]
CommonPool: I'm working in thread Thread[ForkJoinPool.commonPool-worker-1,5,main]
newSingleThreadContext: I'm working in thread Thread[MyOwnThread,5,main]
context: I'm working in thread Thread[main,5,main]
```

从上面的结果可以看出：

- 使用无限制的 Unconfined 上下文的协程运行在主线程中。
- 继承了 runBlocking {…} 的 context 的协程继续在主线程中执行。
- 而 CommonPool 在 ForkJoinPool.commonPool 中。
- 使用 newSingleThreadContext 函数新建的协程上下文，该协程运行在自己的新线程 Thread[MyOwnThread, 5, main] 中。

另外，我们还可以在使用 runBlocking 的时候显式地指定上下文，同时使用 run 函数来更改协程的上下文：

```
fun log(msg: String) = println("${Thread.currentThread()} $msg")

    fun testRunBlockingWithSpecifiedContext() = runBlocking {
    log("$context")
    log("${context[Job]}")
    log("开始")
```

```
    val ctx1 = newSingleThreadContext("线程A")
    val ctx2 = newSingleThreadContext("线程B")
    runBlocking(ctx1) {
        log("Started in Context1")
        run(ctx2) {
            log("Working in Context2")
        }
        log("Back to Context1")
    }
    log("结束")
}
```

运行输出：

```
Thread[main,5,main] [BlockingCoroutine{Active}@b1bc7ed, EventLoopImpl@7cd84586]
Thread[main,5,main] BlockingCoroutine{Active}@b1bc7ed
Thread[main,5,main] 开始
Thread[线程A,5,main] Started in Context1
Thread[线程B,5,main] Working in Context2
Thread[线程A,5,main] Back to Context1
Thread[main,5,main] 结束
```

9.4.2 父子协程

在计算机科学中，例程（Routine）（过程、函数、方法、子程序）被定义为操作的序列。例程的执行形成了父子关系，且孩子例程总是在父例程前结束。协程是例程的泛化概念。

当我们使用协程A的上下文启动另一个协程B时，A就是B的父协程，B将成为A的子协程。当父协程A任务被取消时，B以及它的所有子协程都会被递归地取消。代码示例如下：

```
fun testChildrenCoroutine()= runBlocking<Unit> {
    val request = launch(CommonPool) {
        log("ContextA1: ${context}")

        val job1 = launch(CommonPool) {
            println("job1: 独立的协程上下文!")
            delay(1000)
            println("job1: 不会受到request.cancel()的影响")
        }
        //继承父上下文: request的context
        val job2 = launch(context) {
            log("ContextA2: ${context}")
            println("job2: 是request coroutine的子协程")
            delay(1000)
            println("job2: 当request.cancel(), job2也会被取消")
        }
        job1.join()
        job2.join()
    }
```

```
    delay(500)
    request.cancel()
    delay(1000)
    println("main: Who has survived request cancellation?")
}
```

运行输出：

```
Thread[ForkJoinPool.commonPool-worker-1,5,main] ContextA1: [StandaloneCoroutine
    {Active}@5b646af2, CommonPool]
job1: 独立的协程上下文！
Thread[ForkJoinPool.commonPool-worker-3,5,main] ContextA2: [StandaloneCoroutine
    {Active}@75152aa4, CommonPool]
job2: 是 request coroutine 的子协程
job1: 不会受到 request.cancel() 的影响
main: Who has survived request cancellation?
```

9.5　通道与管道

延迟对象提供了一种在协程之间传输单个值的方法。而通道（Channel）提供了一种传输数据流的方法。通道是使用 SendChannel 和使用 ReceiveChannel 之间的非阻塞通信。

9.5.1　通道 vs 阻塞队列

通道的概念类似于阻塞队列（BlockingQueue）。在 Java 的 Concurrent 包中，BlockingQueue 很好地解决了多线程中如何高效安全“传输”数据的问题。它有两个常用的方法如下：

- E take()：取走 BlockingQueue 里排在首位的对象，若 BlockingQueue 为空，阻塞进入等待状态直到 BlockingQueue 有新的数据被加入；
- put(E e)：把对象 e 加到 BlockingQueue 里，如果 BlockQueue 没有空间，则调用此方法的线程被阻塞，直到 BlockingQueue 里面有空间再继续。

通道与阻塞队列的关键区别是：通道有挂起的操作而不是阻塞的，同时它可以关闭。

代码示例：

```
package com.easy.kotlin

import kotlinx.coroutines.experimental.CommonPool
import kotlinx.coroutines.experimental.channels.Channel
import kotlinx.coroutines.experimental.launch
import kotlinx.coroutines.experimental.runBlocking

/**
 * Created by jack on 2017/7/13.
 */
class ChannelsDemo {
    fun testChannel() = runBlocking<Unit> {
```

```
        val channel = Channel<Int>()
        launch(CommonPool) {
            for (x in 1..10) channel.send(x * x)
        }
        println("channel = ${channel}")
        //here we print five received integers:
        repeat(10) { println(channel.receive()) }
        println("Done!")
    }
}

fun main(args: Array<String>) {
    val cd = ChannelsDemo()
    cd.testChannel()
}
```

运行输出：

```
channel = kotlinx.coroutines.experimental.channels.RendezvousChannel@2e817b38
1
4
9
16
25
36
49
64
81
100
Done!
```

可以看出，使用 Channel<Int>() 背后调用的是会合通道 RendezvousChannel()，会合通道中没有任何缓冲区。send 函数被挂起直到另外一个协程调用 receive 函数，然后 receive 函数挂起直到另外一个协程调用 send 函数。它是一个完全无锁的实现。

9.5.2 关闭通道和迭代遍历元素

通道与队列不同，通道可以关闭，以指示没有更多的元素。在接收端，可以使用 for 循环从通道接收元素。代码示例如下：

```
fun testClosingAndIterationChannels() = runBlocking {
    val channel = Channel<Int>()
    launch(CommonPool) {
        for (x in 1..5) channel.send(x * x)
        channel.close() //我们结束 sending
    }
    //打印通道中的值，直到通道关闭
    for (x in channel) println(x)
```

```
    println("Done!")
}
```

其中，close 函数在这个通道上发送一个特殊的“关闭令牌”。这是一个幂等运算：对此函数的重复调用不起作用，并返回“false”。此函数执行后，isClosedForSend 返回“true”。但是，ReceiveChannel 的 isClosedForReceive 在所有之前发送的元素收到之后才返回“true”。

我们把上面的代码加入打印日志：

```
fun testClosingAndIterationChannels() = runBlocking {
    val channel = Channel<Int>()
    launch(CommonPool) {
        for (x in 1..5) {
            channel.send(x * x)
        }
        println("Before Close => isClosedForSend = ${channel.isClosedForSend}")
        channel.close() // 我们结束 sending
        println("After Close => isClosedForSend = ${channel.isClosedForSend}")
    }
    // 打印通道中的值，直到通道关闭
    for (x in channel) {
        println("${x} => isClosedForReceive = ${channel.isClosedForReceive}")
    }
    println("Done!  => isClosedForReceive = ${channel.isClosedForReceive}")
}
```

运行输出：

```
1 => isClosedForReceive = false
4 => isClosedForReceive = false
9 => isClosedForReceive = false
16 => isClosedForReceive = false
25 => isClosedForReceive = false
Before Close => isClosedForSend = false
After Close => isClosedForSend = true
Done!  => isClosedForReceive = true
```

9.5.3 生产者 - 消费者模式

使用协程生成元素序列的模式非常常见。这是在并发代码中经常有的生产者 - 消费者模式。代码示例：

```
fun produceSquares() = produce<Int>(CommonPool) {
    for (x in 1..7) send(x * x)
}

fun consumeSquares() = runBlocking{
    val squares = produceSquares()
    squares.consumeEach { println(it) }
    println("Done!")
}
```

这里的 produce 函数定义如下：

```
public fun <E> produce(
    context: CoroutineContext,
    capacity: Int = 0,
    block: suspend ProducerScope<E>.() -> Unit
): ProducerJob<E> {
    val channel = Channel<E>(capacity)
    return ProducerCoroutine(newCoroutineContext(context), channel).apply {
        initParentJob(context[Job])
        block.startCoroutine(this, this)
    }
}
```

其中，参数说明如下：

参数名	说　明
context	协程上下文
capacity	通道缓存容量大小（默认没有缓存）
block	协程代码块

produce 函数会启动一个新的协程，协程中发送数据到通道生成数据流，并以 ProducerJob 对象返回对协程的引用。ProducerJob 继承了 Job，为 ReceiveChannel 类型。

9.5.4 使用管道生产无限序列

管道（pipeline）是一个允许单向信息传递的数据结构。从管道“写入端”（SendChannel）写入的数据可以从“读取端”（ReceiveChannel）读回。管道是串行的；从管道中读取的数据总是会保持写入时的顺序。一般来说，管道通常用于一个进程中两个线程之间的通信，或用于父子进程之间的通信。在协程中，同样存在管道这样的数据结构。

我们可以用一个管道协程任务来生产一个无限序列：

```
fun produceNumbers() = produce<Long>(CommonPool) {
    var x = 1L
    while (true) send(x++) // infinite stream of integers starting from 1
}
```

消费序列的函数如下：

```
fun produceNumbers() = produce<Long>(CommonPool) {
    var x = 1L
    while (true) send(x++) // infinite stream of integers starting from 1
}
```

主代码启动并连接整个管线：

```
fun testPipeline() = runBlocking {
    val numbers = produceNumbers() // produces integers from 1 and on
```

```
    val squares = consumeNumbers(numbers) // squares integers
    //for (i in 1..6) println(squares.receive())
    while (true) {
        println(squares.receive())
    }
    println("Done!")
    squares.cancel()
    numbers.cancel()
}
```

运行上面的代码，我们将会发现控制台在打印一个无限序列，完全没有停止的意思。

9.5.5 管道与无穷质数序列

我们使用协程管道来生成一个无穷质数序列。

从无穷大的自然数序列开始：

```
fun numbersProducer(context: CoroutineContext, start: Int) = produce<Int>(context) {
    var n = start
    while (true) send(n++) // infinite stream of integers from start
}
```

这次我们引入一个显式上下文参数 context，以便调用方可以控制协程运行的位置。

下面的管道将筛选传入的数字流，过滤掉可以被当前质数整除的所有数字：

```
fun filterPrimes(context: CoroutineContext, numbers: ReceiveChannel<Int>,
    prime: Int) = produce<Int>(context) {
    for (x in numbers) if (x % prime != 0) send(x)
}
```

现在我们通过 2 开始，从当前通道中取一个质数，并为找到的每个质数启动新的管道阶段，从而构建出管道：

```
numbersFrom(2) -> filterPrimes(2) -> filterPrimes(3) -> filterPrimes(5) ->
    filterPrimes(7) ...
```

测试无穷质数序列：

```
fun producePrimesSequences() = runBlocking {
    var producerJob = numbersProducer(context, 2)

    while (true) {
        val prime = producerJob.receive()
        print("${prime} \t")
        producerJob = filterPrimes(context, producerJob, prime)
    }
}
```

运行上面的代码，我们将会看到控制台一直在无限地打印出质数序列：

```
2    3    5    7    11   13   17   19   23   29   31   37
41   43   47   53   59   61   67   71   73   79   83   89
97   101  103  107  109  113  127  131  137  139  149  151
157  163  167  173  179  181  191  193  197  199  211 …
```

9.5.6 通道缓冲区

我们可以给通道设置一个缓冲区：

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val channel = Channel<Int>(4)          //创建一个缓冲区容量为 4 的通道
    launch(context) {
        repeat(10) {
            println("Sending $it")
            channel.send(it)               //当缓冲区已满的时候，send 将会挂起
        }
    }
    delay(1000)
}
```

输出：

```
Sending 0
Sending 1
Sending 2
Sending 3
Sending 4
```

9.5.7 构建无穷惰性序列

我们可以使用 buildSequence 序列生成器构建一个无穷惰性序列：

```
val fibonacci = buildSequence {
    yield(1L)
    var current = 1L
    var next = 1L
    while (true) {
        yield(next)
        val tmp = current + next
        current = next
        next = tmp
    }
}
```

我们通过 buildSequence 创建一个协程，生成一个惰性的无穷斐波那契数列。该协程通过调用 yield() 函数来产生连续的斐波纳契数。

我们可以从该序列中取出任何有限的数字列表，例如：

```
println(fibonacci.take(16).toList())
```

结果如下：

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

9.6　协程与线程

协程并不是取代线程，而是抽象于线程之上。线程是被分割的 CPU 资源，协程是组织好的代码流程。协程需要线程来承载运行，线程是协程的资源，但协程不会直接使用线程。协程直接利用的是执行器（Interceptor），执行器可以关联任意线程或线程池。线程是协程的资源。协程通过 Interceptor 间接使用线程这个资源。

本节通过对比来简单介绍一下协程与线程的关系与区别，然后讲一下使用协程的好处。

9.6.1　轻量级的协程

直接运行下面的代码：

```
fun testThread() {
    val jobs = List(100_1000) {
        Thread({
            Thread.sleep(1000L)
            print(".")
        })
    }
    jobs.forEach { it.start() }
    jobs.forEach { it.join() }
}
```

我们应该会看到输出报错：

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new
    native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:714)
    at com.easy.kotlin.LightWeightCoroutinesDemo.testThread
        (LightWeightCoroutinesDemo.kt:30)
    at com.easy.kotlin.LightWeightCoroutinesDemoKt.main
        (LightWeightCoroutinesDemo.kt:40)
.....................................................................
```

这里直接启动了 100 000 个线程，并 join 到一起打印“.”，不出意外的我们收到了 java.lang.OutOfMemoryError 异常。这个异常问题本质原因是我们创建了太多的线程，而能创建的线程数是有限制的，导致了异常的发生。在 Java 中，当我们创建一个线程的时候，虚拟机会在 JVM 内存创建一个 Thread 对象同时创建一个操作系统线程，而这个系统线程的内存用的不是 JVMMemory，而是系统中剩下的内存（MaxProcessMemory-JVMMemory-ReservedOsMemory）。能创建的线程数的具体计算公式如下：

Number of Threads = (MaxProcessMemory – JVMMemory – ReservedOsMemory) / (ThreadStackSize)

其中，参数说明如下：

参　数	说　明
MaxProcessMemory	指的是一个进程的最大内存。
JVMMemory	JVM 内存。
ReservedOsMemory	保留的操作系统内存。
ThreadStackSize	线程栈的大小。

我们通常在优化这种问题的时候，要么是采用减小 Thread Stack Size 的方法，要么是采用减小 heap 或 permgen 初始分配的大小方法等方式来临时解决问题。

在协程中，情况完全就不一样了。我们看一下实现上面的逻辑的协程代码：

```
fun testLightWeightCoroutine() = runBlocking {
    val jobs = List(100_000) {
        // create a lot of coroutines and list their jobs
        launch(CommonPool) {
            delay(1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() } // wait for all jobs to complete
}
```

运行上面的代码，我们将看到输出：

```
START: 21:22:28.913
.....................
.....................(100000个)
.....END: 21:22:30.956
```

上面的程序在 2s 左右的时间内正确执行完毕。

9.6.2 协程 vs 守护线程

在 Java 中有两类线程：用户线程（User Thread）和守护线程（Daemon Thread）。

所谓守护线程是指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾回收线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因此，当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。

我们来看一段 Thread 的守护线程的代码：

```
fun testDaemon2() {
    val t = Thread({
        repeat(100) { i ->
            println("I'm sleeping $i ...")
            Thread.sleep(500L)
```

```
        }
    })
    t.isDaemon = true     //必须在启动线程前调用，否则会报错：Exception in thread
                          //"main" java.lang.IllegalThreadStateException
    t.start()
    Thread.sleep(2000L)  //延迟后退出
}
```

这段代码启动一个线程，并设置为守护线程。线程内部是间隔500ms重复打印100次输出。外部主线程睡眠2s。

运行这段代码，将会输出：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
I'm sleeping 3 ...
```

协程跟守护线程很像，用协程来写上面的逻辑，代码如下：

```
fun testDaemon1() = runBlocking {
    launch(CommonPool) {
        repeat(100) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
    delay(2000L) //just quit after delay
}
```

运行这段代码，我们发现也输出：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
I'm sleeping 3 ...
```

我们可以看出，活动的协程不会使进程保持活动状态，它们的行为就像守护程序线程。

9.6.3 区别

协程是编译器级的，而线程是操作系统级的。

协程通常是由编译器来实现的机制。线程看起来也在语言层次，但是内在原理却是操作系统先有这个东西，然后通过一定的API暴露给用户使用，两者在这里有不同。

协程就是用户空间下的线程。用协程来做的东西，用线程或进程通常也是一样可以做的，但往往多了许多加锁和通信的操作。

线程是抢占式，而协程是非抢占式的，所以需要用户自己释放使用权来切换到其他协程，因此同一时间其实只有一个协程拥有运行权，相当于单线程的能力。

9.6.4 协程的好处

协程的优势如下：

- CPU 消耗低。与多线程、多进程等并发模型不同，协程依靠用户空间调度，而线程、进程则是依靠内核来进行调度。因此线程、进程间切换都需要从用户态进入内核态。而协程的切换完全是在用户态完成，程序只在用户空间内切换上下文，不再陷入内核来做线程切换，这样可以避免大量的用户空间和内核空间之间的数据拷贝，降低了 CPU 的消耗，从而大大减缓高并发场景时 CPU 瓶颈的窘境。
- 简化了多线程同步的复杂性。通常多个运行在同一调度器中的协程运行在一个线程内，这也消除掉了多线程同步等带来的编程复杂性。同一时刻同一调度器中的协程只有一个会处于运行状态。使用协程可以很简单地实现一个随时中断随时恢复的函数。Kotlin 的协程库底层封装了丰富的协程函数，大大简化了并发编程的复杂度。
- 摆脱异步编程的一堆 callback 函数。使用协程，我们不再需要像异步编程时写那么一堆 callback 函数，代码结构不再支离破碎，整个代码逻辑上看上去和同步代码没什么区别，简单、易理解、优雅。

9.6.5 Kotlin 协程的内部机制

1. Continuation 对象

Continuation 是一种描述程序的控制状态的抽象，它用一个数据结构来表示一个执行到指定位置的计算过程；这个数据结构可以由程序语言访问而不是隐藏在运行时环境中。

Continuation 这个概念就协程来说就是协程保护的现场。而对于函数来说就是保存函数调用现场——Stack Frame 值和寄存器，以供以后调用继续从 Continuation 处执行。换一个角度看，它也可以看作是非结构化 Goto 语句的函数表达。当我们执行 Yield 从协程返回的时候，需要保存的就是 Continuation 了。

Continuation 在生成之后可作为控制结构使用，在调用时会从它所表示的控制点处恢复执行。One-shot continuation 实现中，控制栈由栈段（Stack segment）组成的链表来表示，整个控制栈被构造成栈帧（Stack of frame）或活动记录（Activation record）。捕获 Continuation 时，当前栈段被保存到 Continuation 中，然后分配一个新的栈段。调用 Continuation 时，丢弃当前栈段，控制返回到之前保存的栈段。

创建一个协程同样包括分配一个单独的栈，但挂起和恢复协程的代价只比标准的函数调用略高。

Kotlin 的协程完全通过编译技术实现（不需要来自 VM 或 OS 端的支持）。挂起机制是通过状态机来实现，其中的状态对应于挂起调用。在挂起时，对应的协程状态与局部变量等一起被存储在编译器生成的类的字段中。在恢复该协程时，恢复局部变量并且状态机从挂起点接着后面的状态继续执行。挂起的协程是作为 Continuation 对象来存储和传递，Continuation 中持有协程挂起状态与局部变量。

关于协程工作原理的更多细节可以在这个设计文档中找到：https://github.com/Kotlin/kotlin-coroutines/blob/master/kotlin-coroutines-informal.md。

2. Kotlin 中协程的标准 API

协程有三个主要组成部分：

- 语言支持（即如上所述的挂起功能）。
- Kotlin 标准库中的底层核心 API。
- 可以直接在用户代码中使用的高级 API。

其中，底层 API（kotlin.coroutines）相对较小，它主要用来实现库级代码，由两个主要包组成：

- kotlin.coroutines.experimental 带有主要类型与下述原语：

```
createCoroutine()
startCoroutine()
suspendCoroutine()
```

- kotlin.coroutines.experimental.intrinsics 带有更底层的函数：

```
suspendCoroutineOrReturn
```

另外，kotlinx.coroutines 库中包含了大多数的应用程序级 API。这个库主要包括下面几大模块：

- 使用 kotlinx-coroutines-core 进行平台无关的异步编程。
- 基于 JDK 8 中的 CompletableFuture 的 API：kotlinx-coroutines-jdk8。
- 基于 JDK 7 及更高版本 API 的非阻塞 IO（NIO）：kotlinx-coroutines-nio。
- 支持 Swing（kotlinx-coroutines-swing）和 JavaFx（kotlinx-coroutines-javafx）。
- 支持 RxJava：kotlinx-coroutines-rx。

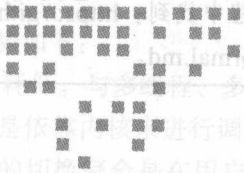
关于这些 API 用法的更多细节可以参考相关文档。

9.7　本章小结

本章通过大量实例介绍了协程的用法，以及作为轻量级线程的协程是怎样简化多线程并发编程的。我们看到，协程通过挂起机制实现非阻塞的特性大大提升了并发性能。最后，还简单介绍了协程的实现原理以及标准 API 库。Kotlin 的协程实现大量地调用了 Java 中的多线程 API，所以在 Kotlin 中，我们仍然完全可以使用 Java 中的多线程编程。

下一章我们来一起学习 Kotlin 与 Java 代码之间的互相调用。

本章示例代码：https://github.com/EasyKotlin/chapter9_coroutines



Chapter 10 第 10 章

Kotlin 与 Java 互操作

Kotlin 与 Java、Android 是 100% 互操作的。

在前面的章节中，我们已经学习了 Kotlin 的基础语法、类型系统、泛型与集合类、面向对象与函数式编程等主题，在上一章中我们还看到了 Kotlin 提供的轻量级并发编程模型：协程的相关内容。从本章开始我们将进入工程代码的实战。我们将在后面分别介绍 Kotlin 集成 Spring Boot 开发服务端 Web 项目、使用 Kotlin 开发 Android 项目，以及使用 Kotlin 来写前端 JavaScript 代码等主题。

Kotlin 的竞争优势在于它并不是完全隔离于 Java 语言，而是可与 Java 100% 互操作的。这样，Kotlin 就可以站在整个 Java 生态巨人的肩上，向着更远大的前程前进。

本章我们就让我们一起来学习下 Kotlin 与 Java 的互操作，主要内容包括 Kotlin 调用 Java 的集合类、getter/setter、null 对象、Object 对象方法、数组与可变参数、静态成员、SAM 等；还包括 Java 调用 Kotlin 的类属性、包级函数、实例字段、静态字段 / 方法等。同时还介绍了互操作中关键字冲突的处理、Java 跟 Kotlin 中异常的不同以及互操作、Java 与 Kotlin 的类型映射。最后一小节中我们通过丰富的实例简单对比了 Kotlin 与 Java 的异同，从中可以看到 Kotlin 相比于 Java 的极简优雅风格。

10.1 在 Kotlin 中调用 Java

Kotlin 很像 Java，它长得不像 Clojure 或者 Scala 那么奇怪（承认现实吧，这两种语言就是挺奇怪的），所以 Java 程序员学 Kotlin 应该很快，这门语言显然就是写给 Java 开发者

来用的。

Kotlin 在设计之初就考虑了与 Java 的互操作性。我们可以从 Kotlin 中自然地调用现成的 Java 代码。下面从示例开始讲解。

10.1.1　调用 Okhttp 库示例

例如，下面是一个 Kotlin 调用 Java 中的 Okhttp 库的代码：

```
package com.easy.kotlin

import okhttp3.*
import java.io.File
import java.io.IOException
import java.util.concurrent.TimeUnit

object OkhttpUtils {
    fun get(url: String): String? {
        var result: String? = ""
        val okhttp = OkHttpClient.Builder()
                .connectTimeout(1, TimeUnit.HOURS)
                .readTimeout(1, TimeUnit.HOURS)
                .writeTimeout(1, TimeUnit.HOURS)
                .build()

        val request = Request.Builder()
                .url(url)
                .build()

        val call = okhttp.newCall(request)

        try {
            val response = call.execute()
            result = response.body()?.string()
            val f = File("run.log")
            f.appendText(result!!)
            f.appendText("\n")

        } catch (e: IOException) {
            e.printStackTrace()
        }

        return result
    }
}
```

Kotlin 调用 Java 代码跟 Groovy 一样流畅自如，但是不像 Groovy 那样“怎么写都对，但是一运行就报错”，因为 Groovy 是一门动态类型语言，而 Kotlin 是一门强类型的静态类型语言。我们基本不需要改变什么就可以直接使用 Java 中的 API 库。

并且在 Java 代码中也可以很顺利地调用 Kotlin 代码，如下所示：

```
package com.easy.kotlin;

import com.alibaba.fastjson.JSON;

/**
 * Created by jack on 2017/7/14.
 */
public class JSONUtils {
    public static String toJsonString(Object o) {
        return JSON.toJSONString(o);
    }

    public static void main(String[] args) {
        String url = "http://www.baidu.com";
        String result = OkhttpUtils.INSTANCE.get(url);
        System.out.println(result);
    }
}
```

因为 Kotlin 跟 Java 本是两门语言，所以在互相调用的时候，会有一些特殊的语法。这里使用 Java 调用 Kotlin 的 object 对象函数的语法就是 OkhttpUtils.INSTANCE.get(url)，我们看到这里多了个 INSTANCE。

我们甚至也可以在一个项目中同时使用 Kotlin 和 Java 两种语言混合编程。我们在下一章中将会看到，在一个 Spring Boot 工程中同时使用了 Kotlin 和 Java 两种语言进行混合开发。

下面我们来继续介绍 Kotlin 调用 Java 代码的一些细节。

10.1.2 使用 Java 的集合类

Kotlin 的集合类 API 很多就是直接使用 Java 的 API 来实现的。我们在使用的时候，毫无违和感，自然天成。示例代码如下：

```
@RunWith(JUnit4::class)
class KotlinUsingJavaTest {
    @Test fun testArrayList() {
        val source = listOf<Int>(1, 2, 3, 4, 5)
        //使用 Java 的 ArrayList
        val list = ArrayList<Int>()
        for (item in source) {
            list.add(item) //ArrayList.add()
        }
        for (i in 0..source.size - 1) {
            list[i] = source[i] //调用 get 和 set
        }
    }
}
```

10.1.3　调用 Java 中的 Getter 和 Setter

在 Java 中遵循这样的约定：Getter 方法无参数并以 get 开头，Setter 方法单参数并以 set 开头。在 Kotlin 中可以直接表示为属性。例如，我们写一个带 Setter 和 Getter 的 Java 类如下所示：

```
package com.easy.kotlin;

import java.util.Date;

/**
 * Created by jack on 2017/7/14.
 */
public class Product {
    Long id;
    String name;
    String category;
    Date gmtCreated;
    Date gmtModified;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public Date getGmtCreated() {
        return gmtCreated;
    }

    public void setGmtCreated(Date gmtCreated) {
```

```
        this.gmtCreated = gmtCreated;
    }

    public Date getGmtModified() {
        return gmtModified;
    }

    public void setGmtModified(Date gmtModified) {
        this.gmtModified = gmtModified;
    }
}
```

然后，我们在 Kotlin 可以直接使用属性名字进行 get 和 set 操作：

```
@RunWith(JUnit4::class)
class ProductTest {
    @Test fun testGetterSetter() {
        val product = Product()
        product.name = "账务系统"
        product.category = "金融财务类"
        product.gmtCreated = Date()
        product.gmtModified = Date()
        println(JSONUtils.toJsonString(product))
        Assert.assertTrue(product.getName() == "账务系统")
        Assert.assertTrue(product.name == "账务系统")
        Assert.assertTrue(product.getCategory() == "金融财务类")
        Assert.assertTrue(product.category == "金融财务类")
    }
}
```

当然，我们还可以像在 Java 中一样，直接调用 product.getName()、product.setName（“Kotlin”）等 Getter、Setter 方法。

10.1.4 调用 Java 中返回 void 的方法

如果一个 Java 方法返回 void，那么 Kotlin 调用该方法时，返回 Unit：

```
public class Admin {
    String name;

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Admin{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

我们这样调用：

```
val setReturn = admin.setName("root")
println(setReturn)
```

将输出：

```
kotlin.Unit
```

10.1.5　调用来自 Java 中的 null

我们知道 Java 中的任何引用都可能是 null，这样在使用 Kotlin 调用来自 Java 的对象的时候就有可能会出现空安全的问题。下面分两个方面来分析这个问题如何解决。

1. 平台类型

Java 声明的类型在 Kotlin 中会被特别对待并称为“平台类型”（platform type），对这种类型的空检查会放宽，因此它们的安全保证与在 Java 中相同。

请看以下示例：

```
@RunWith(JUnit4::class)
class CallingJavaNullSafe {
    @Test fun testCallingJavaNullSafe() {
        val product = Product()
        //product.name = null
        product.category = "金融财务类"
        product.gmtCreated = Date()
        product.gmtModified = Date()
        println(JSONUtils.toJsonString(product))

        val name = product.name
        println("product name is ${name}")

        val eqName = name == "账务系统"
        println(eqName)

        name.substring(1)
    }
}
```

上面的代码可以正确编译通过。Kotlin 编译器对来自 Java 的空值 name（平台类型）放宽了空检查 name.substring(1)。但是这样的空指针异常仍然会在运行时抛出来。

运行上面的代码，我们可以看到以下输出：

```
{"category":"金融财务类","gmtCreated":1500050426817,"gmtModified":1500050426817}
product name is null
false

null cannot be cast to non-null type java.lang.String
```

```
kotlin.TypeCastException: null cannot be cast to non-null type java.lang.String
    at com.easy.kotlin.CallingJavaNullSafe.testCallingJavaNullSafe
        (CallingJavaNullSafe.kt:27)
```

我们没有设置 name 的值，在 Java 中，它就是 null。我们在 Kotlin 代码中使用了这个 name 进行计算，我们可以看出：

```
val eqName = name == "账务系统"
println(eqName)
```

可以正确输出 false。这表明 Kotlin 的判断字符串是否相等已经对 null 的情况作了判断处理，这样的代码如果在 Java 中调用 name.equals（"账务系统"）就该抛出空指针异常了。

但是当我们直接使用 name 这个值来调用 name.substring(1) 的时候，Kotlin 编译器不会检查这个空异常，且运行时还是要报错的：null cannot be cast to non-null type java.lang.String。

如果我们不想看到这样的异常，而是当 name 是 null 的时候，安静地输出 null，直接使用 Kotlin 中的空安全的调用 .?：

```
name?.substring(1)
```

这样，运行的时候不会抛出异常，直接安静地返回 null。

2. 平台类型助记符

平台类型不能在程序中显式表述，因此在语言中没有相应语法。然而，编译器和 IDE 有时需要（在错误信息中、参数信息中等）显示它们，所以我们用助记符来表示：

- T!：表示 T 或者 T?。
- (Mutable) Collection<T>!：表示“可变或不可变、可空或不可空的 T 的 Java 集合”。
- Array<(out) T>!：表示可空或者不可空的 T（或 T 的子类型）的 Java 数组。

10.1.6 使用 Java 的泛型

Kotlin 的泛型与 Java 有点不同。当将 Java 类型导入 Kotlin 时，会执行一些转换，参见表 10-1。

表 10-1 Kotlin 的泛型与 Java 的泛型比较

Kotlin 的泛型	Java 的泛型	说 明
Foo<out Bar!>!	Foo<? extends Bar>	Java 的通配符转换成类型投影
Foo<? super Bar>	Foo<in Bar!>!	Java 的通配符转换成类型投影
List<*>!	List	Java 的原始类型转换成星投影

与 Java 一样，Kotlin 在运行时不保留泛型，即对象不携带传递到它们构造器中的那些类型参数的实际类型。即 ArrayList<Integer>() 和 ArrayList<Character>() 是不能区分的。

10.1.7　数组的互操作

与 Java 不同，Kotlin 中的数组是非型变的，即 Kotlin 不允许我们把一个 Array<String> 赋值给一个 Array<Any>。

在 Java 平台上，持有原生数据类型的数组避免了装箱 / 拆箱操作的开销。

在 Kotlin 中，对于每种原生类型的数组都有一个特化的类（IntArray、DoubleArray、CharArray 等）来实现同样的功能。它们与 Array 类无关，并且会编译成 Java 原生类型数组以获得最佳性能。

10.1.8　可变参数的互操作

Java 类有时声明一个具有可变数量参数（varargs）的方法来使用索引：

```
public class VarArgsDemo<T> {
    static VarArgsDemo vad = new VarArgsDemo();

    public static void main(String... agrs) {
        System.out.println(vad.append("a", "b", "c"));
        System.out.println(vad.append(1, 2, 3));
        System.out.println(vad.append(1, 2, "3"));
    }

    public String append(T... element) {
        StringBuilder result = new StringBuilder();
        for (T e : element) {
            result.append(e);
        }
        return result.toString();
    }
}
```

在 Kotlin 中，我们使用展开运算符 * 来传递这个 varargs：

```
@RunWith(JUnit4::class)
class VarArgsDemoTest {
    @Test fun testVarArgsDemo() {
        val varArgsDemo = VarArgsDemo<Any?>()
        val array = arrayOf(0, 1, 2, 3)
        val result = varArgsDemo.append(*array)
        println(result)
    }
}
```

运行输出：

```
0123
```

10.1.9　检查 Java 中的异常

在 Kotlin 中，所有异常都是非受检的（Non-Checked Exception），这意味着编译器不会

强迫你捕获其中的任何一个。而在 Java 中会要求我们捕获异常，例如下面的代码：

```
public class JSONUtils {

    static JSONUtils jsonUtils = new JSONUtils();

    public static String toJsonString(Object o) {
        return JSON.toJSONString(o);
    }

    public static void main(String[] args) {
        String url = "http://www.baidu.com";
        String result = OkhttpUtils.INSTANCE.get(url);
        System.out.println(result);

        jsonUtils.parseObject( jsonText: "{}");
    }
    Unhandled exception: java.lang.Exception

    public Object parseObject(String jsonText) throws Exception {
        throw new Exception("Test Exception");
    }
}
```

也就是说，我们需要写类似下面的 try catch 代码块：

```
try {
    jsonUtils.parseObject("{}");
} catch (Exception e) {
    e.printStackTrace();
}
```

然而在 Kotlin 中情况就不是这样子了：当我们调用一个声明受检异常的 Java 方法时，Kotlin 不会强迫你做任何事情：

```
@Test fun testNonCheckedExceptions() {
    val jsonUtils = JSONUtils()
    jsonUtils.parseObject("{}")
}
```

但是，我们在运行的时候，还是会抛出异常：

```
com.easy.kotlin.CallingJavaNullSafe > testNonCheckedExceptions FAILED
    java.lang.Exception at CallingJavaNullSafe.kt:34
```

Kotlin 的不受检异常会导致运行时抛出异常。异常需要处理，该处理的终归还是要处理的。

10.1.10 调用 Java 的 Object 方法

Java 中的 java.lang.Object 定义如下：

```
public class Object {
    private static native void registerNatives();
```

```
static {
    registerNatives();
}
public final native Class<?> getClass();
public native int hashCode();
public boolean equals(Object obj) {
    return (this == obj);
}
protected native Object clone() throws CloneNotSupportedException;
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
public final native void notify();
public final native void notifyAll();
public final native void wait(long timeout) throws InterruptedException;
public final void wait(long timeout, int nanos) throws InterruptedException {...}
public final void wait() throws InterruptedException {
    wait(0);
}
protected void finalize() throws Throwable { }
```

当 Java 类型导入到 Kotlin 中时，类型 java.lang.Object 的所有引用都成了 Any。Any 只声明了 toString()、hashCode() 和 equals() 函数。怎样才能用到 java.lang.Object 的其他成员方法呢？下面我们来看一下。

1. wait()/notify()

《Effective Java》第 69 条中建议优先使用并发工具而不是 wait() 和 notify()。因此，类型 Any 的引用没有提供这两个方法。

如果我们真的需要调用它们，可以将其转换为 java.lang.Object 来使用：

```
(foo as java.lang.Object).wait()
```

2. getClass()

要取得对象的 Java 类，可以在类引用上使用 Java 扩展属性，它是 Kotlin 的反射类 kotlin.reflect.KClass 的扩展属性：

```
val fooClass = foo::class.java
```

上面的代码使用了自 Kotlin 1.1 起支持的绑定类引用。我们也可以使用 javaClass 扩展属性：

```
val fooClass = foo.javaClass
```

3. clone()

要覆盖 clone()，需要继承 kotlin.Cloneable：

```
class Example : Cloneable {
```

```
    override fun clone(): Any { …… }
}
```

要谨慎地改写 clone 方法。

4. finalize()

要覆盖 finalize()，只需要声明它即可，不用再写 override 关键字：

```
class C {
    protected fun finalize() {
        //终止化逻辑
    }
}
```

10.1.11 访问 Java 静态成员

Java 类的静态成员会形成该类的“伴生对象”，我们可以直接显式地访问其成员。例如：一个带静态方法的 Java 类如下：

```
public class JSONUtils {
    public static String toJsonString(Object o) {
        return JSON.toJSONString(o);
    }
}
```

我们在 Kotlin 代码中可以直接这样调用：

```
@RunWith(JUnit4::class)
class JSONUtilsTest {
    @Test fun testJSONUtils() {
        val userService = UserServiceImpl()
        val user = userService.findByName("admin")
        Assert.assertTrue(user.name == "admin")

        val userJson = JSONUtils.toJsonString(user)
        println(userJson)
        Assert.assertTrue(userJson == "{\"name\":\"admin\",\"password\":\"admin\"}")
    }
}
```

上面我们提到过，如果是反过来调用，Java 调用 Kotlin 中的 object 对象类中的函数，需要使用 object 的对象名 .INSTANCE 来调用函数。

10.1.12 使用 Java 的反射

我们可以使用 instance::class.java、ClassName::class.java 或者 instance.javaClass 通过 java.lang.Class 来进入 Java 的反射类 java.lang.Class，之后我们就可以使用 Java 中的反射的功能特性了。

代码示例：

```
@RunWith(JUnit4::class)
class RefectClassTest {
    @Test fun testGetterSetter() {
        val product = Product()
        val pClz = product::class.java
        println(pClz.canonicalName)
        pClz.declaredFields.forEach { println(it) }
        pClz.declaredMethods.forEach {
            println(it.name);
            it.parameters.forEach { println(it) }
        }
    }
}
```

运行上面的代码输出：

```
com.easy.kotlin.Product
java.lang.Long com.easy.kotlin.Product.id
java.lang.String com.easy.kotlin.Product.name
java.lang.String com.easy.kotlin.Product.category
java.util.Date com.easy.kotlin.Product.gmtCreated
java.util.Date com.easy.kotlin.Product.gmtModified
getName
setName
java.lang.String arg0
getId
setId
java.lang.Long arg0
setCategory
java.lang.String arg0
getGmtCreated
setGmtCreated
java.util.Date arg0
getGmtModified
setGmtModified
java.util.Date arg0
getCategory
```

10.1.13　SAM 转换

我们在 Kotlin 中，要某个函数做某件事时，会传一个函数参数给它。而在 Java 中，并不支持传送函数参数。通常 Java 的实现方式是将动作放在一个实现某接口的类中，然后将该类的一个实例传递给另一个方法。在大多数情况下，这些接口只有单个抽象方法（Single Abstract Method），在 Java 中称为 SAM 类型。

例如：Runnable 接口：

```
@FunctionalInterface
```

```
public interface Runnable {
    public abstract void run();
}
```

在 Java 8 中，该接口通常称为函数式接口。

Kotlin 支持 SAM 转换。在 Kotlin 中，我们可以这样调用 Java 的 SAM 接口方法：

```
val runnable = Runnable { println(" 执行测试 ") }
```

测试代码：

```
@RunWith(JUnit4::class)
class SAMFunctionalInterfaceTest {
    @Test fun testSAMFunctionalInterface() {
        val runnable = Runnable { println(" 执行测试 ") }
        val thread = Thread(runnable)
        thread.start()
    }
}
```

要注意的是，SAM 转换只适用于接口，而不适用于抽象类，即使这些抽象类也只有一个抽象方法。

还要注意，此功能只适用于 Java 互操作；因为 Kotlin 具有合适的函数类型，所以不需要将函数自动转换为 Kotlin 接口的实现。

10.2 在 Java 中调用 Kotlin

在 Java 中同样也可以调用 Kotlin 代码，但是要多用一些注解语法。

10.2.1 访问 Kotlin 属性

Kotlin 中的属性字段会编译成以下 Java 元素：

- 一个 Getter 方法，名称通过加前缀 get 算出；
- 一个 Setter 方法，名称通过加前缀 set 算出（只适用于 var 属性）；
- 一个与属性名称相同的私有字段。

例如，下面的 Kotlin 类：

```
class Department {
    var id: Long = -1L
    var name: String = "Dept"
}
```

会被编译成对应的 Java 代码：

```
public final class Department {
    private long id = -1L;
```

```
    @NotNull
    private String name = "Dept";

    public final long getId() {
        return this.id;
    }

    public final void setId(long var1) {
        this.id = var1;
    }

    @NotNull
    public final String getName() {
        return this.name;
    }

    public final void setName(@NotNull String var1) {
        Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
        this.name = var1;
    }
}
```

我们可以看出，在 Kotlin 中的 Long 类型被编译成 Java 中的原生 long 了。

我们在 Java 中这样调用：

```
@RunWith(JUnit4.class)
public class JavaCallingKotlinCodeTest {
    @Test
    public void testProperty() {
        Department d = new Department();
        d.setId(1);
        d.setName("技术部");

        Assert.assertTrue(1 == d.getId());
        Assert.assertTrue("技术部".equals(d.getName()));

    }
}
```

另外，如果 Kotlin 的属性名以 is 开头，则使用不同的名称映射规则：

- Getter 的名称直接使用属性名称。
- Setter 的名称是通过将 is 替换为 set 获得。

例如，对于属性 isOpen，其 Getter 会称为 isOpen()，而其 Setter 会称为 setOpen()。

这一规则适用于任何类型的属性，并不仅限于 Boolean 型。

代码示例如下。

Kotlin 代码：

```
class Department {
```

```
    var id: Long = -1L
    var name: String = "Dept"
    var isOpen:Boolean = true
    var isBig:String = "Y"
}
```

Java 调用 Kotlin 的测试代码：

```
@Test
public void testProperty() {
    Department d = new Department();
    d.setId(1);
    d.setName("技术部");
    d.setBig("Y");
    d.setOpen(true);

    Assert.assertTrue(1 == d.getId());
    Assert.assertTrue("技术部".equals(d.getName()));
    Assert.assertTrue("Y".equals(d.isBig()));
    Assert.assertTrue(d.isOpen());

}
```

10.2.2 调用 Kotlin 的包级函数

在 package com.easy.kotlin 包内的 KotlinExample.kt 源文件中声明的所有函数和属性，包括扩展函数，都将编译成一个名为 com.easy.kotlin.KotlinExampleKt 的 Java 类中的静态方法。

代码示例如下。

Kotlin 的包级属性、函数代码：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/7/15.
 */

fun f1() {
    println("I am f1")
}

fun f2() {
    println("I am f2")
}

val p: String = "PPP"

fun String.swap(index1: Int, index2: Int): String {
    val strArray = this.toCharArray()
```

```
    val tmp = strArray[index1]
    strArray[index1] = strArray[index2]
    strArray[index2] = tmp

    var result = ""
    strArray.forEach { result += it }
    return result
}

fun main(args: Array<String>) {
    println("abc".swap(0, 2))
}
```

编译成对应的 Java 的代码：

```
public final class KotlinExampleKt {
    @NotNull
    private static final String p = "PPP";

    public static final void f1() {
        String var0 = "I am f1";
        System.out.println(var0);
    }

    public static final void f2() {
        String var0 = "I am f2";
        System.out.println(var0);
    }

    @NotNull
    public static final String getP() {
        return p;
    }

    @NotNull
    public static final String swap(@NotNull String $receiver, int index1, int
        index2) {
        Intrinsics.checkParameterIsNotNull($receiver, "$receiver");
        char[] var10000 = $receiver.toCharArray();
        Intrinsics.checkExpressionValueIsNotNull(var10000, "(this as java.
            lang.String).toCharArray()");
        char[] strArray = var10000;
        char tmp = strArray[index1];
        strArray[index1] = strArray[index2];
        strArray[index2] = tmp;
        Object result = "";
        char[] $receiver$iv = strArray;

        for(int var7 = 0; var7 < $receiver$iv.length; ++var7) {
            char element$iv = $receiver$iv[var7];
            result = result + element$iv;
```

```
        }

        return result;
    }

    public static final void main(@NotNull String[] args) {
        Intrinsics.checkParameterIsNotNull(args, "args");
        String var1 = swap("abc", 0, 2);
        System.out.println(var1);
    }
}
```

我们可以看到，Kotlin 中的扩展函数：

```
fun String.swap(index1: Int, index2: Int): String
```

被编译成：

```
public static final String swap(@NotNull String $receiver, int index1, int index2)
```

Kotlin 中的 String 接收者被当作 Java 方法中的第一个参数传入。

Java 调用 Kotlin 包级属性、函数的测试代码：

```
@Test
public void testPackageFun() {
    KotlinExampleKt.f1();
    KotlinExampleKt.f2();
    System.out.println(KotlinExampleKt.getP());
    KotlinExampleKt.swap("abc",0,1);
}
```

运行输出：

```
I am f1
I am f2
PPP
bac
```

另外，要注意这里生成的类 KotlinExampleKt，我们不能使用 new 来创建实例对象：

```
KotlinExampleKt example = new KotlinExampleKt();// 报错
```

报如下错误：

```
error: cannot find symbol
        KotlinExampleKt example = new KotlinExampleKt();
                                      ^
    symbol:   constructor KotlinExampleKt()
    location: class KotlinExampleKt
1 error
```

在编程中，我们推荐使用 Kotlin 默认的命名生成规则。如果确实有特殊场景需要自定

义 Kotlin 包级函数对应的生成 Java 类的名字，可以使用 @JvmName 注解修改生成的 Java 类的类名：

```
@file:JvmName("MyKotlinExample")

package com.easy.kotlin

/**
 * Created by jack on 2017/7/15.
 */

fun f3() {
    println("I am f3")
}

fun f4() {
    println("I am f4")
}

val p2: String = "PPP"
```

测试代码：

```
MyKotlinExample.f3();
MyKotlinExample.f4();
```

10.2.3　调用 Kotlin 中的实例字段

我们使用 @JvmField 注解对 Kotlin 中的属性字段标注，表示这是一个实例字段（Instance Field），Kotlin 编译器在处理的时候，不会给这个字段生成 Getters/Setters 方法：

```
class Department {
    var id: Long = -1L
    var name: String = "Dept"
    var isOpen: Boolean = true
    var isBig: String = "Y"

    @JvmField var NO = 0
}
```

映射成 Java 的代码就是：

```
public final class Department {
    private long id = -1L;
    @NotNull
    private String name = "Dept";
    private boolean isOpen = true;
    @NotNull
    private String isBig = "Y";
    @JvmField
```

```
    public int NO;

    public final long getId() {
        return this.id;
    }

    public final void setId(long var1) {
        this.id = var1;
    }

    @NotNull
    public final String getName() {
        return this.name;
    }

    public final void setName(@NotNull String var1) {
        Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
        this.name = var1;
    }

    public final boolean isOpen() {
        return this.isOpen;
    }

    public final void setOpen(boolean var1) {
        this.isOpen = var1;
    }

    @NotNull
    public final String isBig() {
        return this.isBig;
    }

    public final void setBig(@NotNull String var1) {
        Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
        this.isBig = var1;
    }
}
```

我们在 Java 中调用的时候，就直接使用这个属性实例字段 NO：

```
System.out.println(d.NO = 10);
```

10.2.4 调用 Kotlin 中的静态字段

Kotlin 在命名对象或伴生对象中声明的属性如下：

```
class Department {
    ...
    companion object {
        var innerID = "X001"
```

```
        @JvmField
        var innerName = "DEP"
    }
}
```

innerID 和 innerName 字段的区别在于可见性上：

```
@NotNull
private static String innerID = "X001";
@JvmField
@NotNull
public static String innerName = "DEP";
```

这个私有的 innerID 通过 Companion 对象来封装，提供出 public 的 getInnerID() 和 setInnerID 来访问：

```
public static final class Companion {
    @NotNull
    public final String getInnerID() {
        return Department.innerID;
    }

    public final void setInnerID(@NotNull String var1) {
        Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
        Department.innerID = var1;
    }

    private Companion() {
    }

    // $FF: synthetic method
    public Companion(DefaultConstructorMarker $constructor_marker) {
        this();
    }
}
```

Java 访问 innerID 时候是通过 Companion 进行的：

```
Department.Companion.getInnerID()
```

而我们使用 @JvmField 注解的字段 innerName，Kotlin 编译器会把它的访问权限设置为 public，这样就可以访问这个属性字段了，如下所示：

```
Department.innerName
```

10.2.5　调用 Kotlin 中的静态方法

在 Kotlin 中，还可以将命名对象或伴生对象中定义的函数标注为 @JvmStatic，这样编译器既会在相应对象的类中生成静态方法，也会在对象自身中生成实例方法。

与静态属性类似，我们看下面的代码示例：

```
class Department {
    ...
    companion object {
        var innerID = "X001"
        @JvmField
        var innerName = "DEP"

        fun getObjectName() = "ONAME"
        @JvmStatic
        fun getObjectID() = "OID"
    }
}
```

编译器编译之后，反编译成对应的 Java 代码如下：

```
public final class Department {
    ...
    @JvmStatic
    @NotNull
    public static final String getObjectID() {
        return Companion.getObjectID();
    }

    public static final class Companion {
        ...
        @NotNull
        public final String getObjectName() {
            return "ONAME";
        }

        @JvmStatic
        @NotNull
        public final String getObjectID() {
            return "OID";
        }
        ...
    }
}
```

在 Java 中调用的代码如下：

```
Department.Companion.getObjectID(); //OK
Department.Companion.getObjectName(); //OK, 唯一的工作方式
Department.getObjectID(); //ALSO OK
Department.getObjectName(); //ERROR
```

这些注解语法是编译器为了更加方便 Java 调用 Kotlin 代码提供的一些简便技巧。这样可使得 Java 中调用 Kotlin 代码更加自然优雅些。

10.2.6 Kotlin 与 Java 的可见性

Kotlin 的可见性与 Java 的可见性的映射关系如表 10-2 所示。

表 10-2　Kotlin 可见性与 Java 可见性的映射关系

Kotlin 中的声明	Java 中的声明
private	private
protected	protected
internal	public
public	public

例如下面的 Kotlin 代码：

```
class ProgrammingBook {
    private var isbn: String = "978-7-111-44250-9"
    protected var author: String = "Cay"
    public var name: String = "Core Java"
    internal var pages: Int = 300

    private fun findISBN(): String = "978-7-111-44250-9"
    protected fun findAuthor(): String = "Cay"
    public fun findName(): String = "Core Java"
    internal fun findPages(): Int = 300
}
```

对应的 Java 的代码是：

```
public final class ProgrammingBook {
    private String isbn = "978-7-111-44250-9";
    @NotNull
    private String author = "Cay";
    @NotNull
    private String name = "Core Java";
    private int pages = 300;

    @NotNull
    protected final String getAuthor() {
        return this.author;
    }

    protected final void setAuthor(@NotNull String var1) {
        Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
        this.author = var1;
    }

    @NotNull
    public final String getName() {
        return this.name;
    }

    public final void setName(@NotNull String var1) {
        Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
        this.name = var1;
```

```
    }

    public final int getPages$production_sources_for_module_chapter10_
        interoperability_main() {
        return this.pages;
    }

    public final void setPages$production_sources_for_module_chapter10_
        interoperability_main(int var1) {
        this.pages = var1;
    }

    private final String findISBN() {
        return "978-7-111-44250-9";
    }

    @NotNull
    protected final String findAuthor() {
        return "Cay";
    }

    @NotNull
    public final String findName() {
        return "Core Java";
    }

    public final int findPages$production_sources_for_module_chapter10_
        interoperability_main() {
        return 300;
    }
}
```

可以看到，Kotlin 中的可见性与 Java 中的基本相同。

10.2.7 生成默认参数函数的重载

在 Kotlin 中写一个有默认参数值的 Kotlin 函数，它会对每一个有默认值的参数都生成一个重载函数。在 Java 中调用这样的 Kotlin 函数的话，只有完整参数签名的方法可以调用。例如下面的函数

```
fun f1(a: Int = 0, b: String = "B") {

}
```

在 Java 中只能调用 f1(1, "a") 这一个带两个参数的函数，无法像 Kotlin 中那样可以调用下面 4 个函数：

```
f1(1, "a")
f2()
```

```
f2(2)
ofdp.f2(2, "b")
```

如果想在 Java 代码中也能像在 Kotlin 代码中一样，可以调用 Kotlin 函数自动生成的多个重载函数，那么可以使用 @JvmOverloads 注解。Kotlin 代码示例如下：

```
class OverridesFunWithDefaultParams {
    fun f1(a: Int = 0, b: String = "B") {

    }

    @JvmOverloads fun f2(a: Int = 0, b: String = "B") {

    }
}
```

函数 f1 和 f2 都带有默认参数。测试代码如下：

```
@Test
public void testOverridesFunWithDefaultParams() {
    OverridesFunWithDefaultParams ofdp = new OverridesFunWithDefaultParams();
    ofdp.f1(1, "a");
    ofdp.f2();
    ofdp.f2(2);
    ofdp.f2(2, "b");
}
```

有了 @JvmOverloads 注解，编译器就会针对这个注解所标注的函数进行特殊处理，为之生成额外的重载函数给 Java 调用。

10.2.8　显式声明 Kotlin 中的异常

如上所述，Kotlin 没有受检异常。即像下面像这样的 Kotlin 函数：

```
class CheckKotlinException {
    fun thisIsAFunWithException() {
        throw Exception("I am an exception in kotlin")
    }
}
```

在 Java 中调用，编译器是不会检查这个异常的：

```
@Test
public void testCheckKotlinException() {
    CheckKotlinException cke = new CheckKotlinException();
    cke.thisIsAFunWithException();// Java 编译器不检查这个 Kotlin 中的异常
}
```

当然，在运行时，还是会抛出这个异常。然后，如果我们想要在 Java 中调用它并捕捉这个异常，我们可以给 Kotlin 中的函数加上注解 @Throws(Exception::class)，就像下面这样：

```
@Throws(Exception::class)
fun thisIsAnotherFunWithException() {
    throw Exception("I am Another exception in kotlin")
}
```

在 Java 中调用时，Java 编译器就会检查这个异常：

```
@RunWith(JUnit4.class)
public class CheckKotlinExceptionTest {
    @Test
    public void testCheckKotlinException() {
        CheckKotlinException cke = new CheckKotlinException();
        cke.thisIsAFunWithException();// Java编译器不检查这个Kotlin中的异常

        // Kotlin中显示声明了异常，Java编译器会检查这个异常
        cke.thisIsAnotherFunWithException();

        try {
            cke.thisIsAnotherFunWithException();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Unhandled exception: java.lang.Exception

最后，我们的代码需要捕获该异常并处理它。

完整的示例代码如下：

```
package com.easy.kotlin

/**
 * Created by jack on 2017/7/15.
 */
class CheckKotlinException {
    fun thisIsAFunWithException() {
        throw Exception("I am an exception in kotlin")
    }

    @Throws(Exception::class)
    fun thisIsAnotherFunWithException() {
        throw Exception("I am Another exception in kotlin")
    }
}
```

测试代码：

```
package com.easy.kotlin;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;

/**
```

```
 * Created by jack on 2017/7/15.
 */
@RunWith(JUnit4.class)
public class CheckKotlinExceptionTest {
    @Test
    public void testCheckKotlinException() {
        CheckKotlinException cke = new CheckKotlinException();
        cke.thisIsAFunWithException();// Java 编译器不检查这个 Kotlin 中的异常

        // Kotlin 中显示声明了异常，Java 编译器会检查这个异常
        // cke.thisIsAnotherFunWithException();
        try {
            cke.thisIsAnotherFunWithException();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

10.2.9　关键字冲突的互操作

一些 Kotlin 关键字在 Java 中是有效标识符，如 in、object、is 等。

如果一个 Java 库使用了 Kotlin 关键字作为方法，我们可以通过反引号（`）字符转义它来调用该方法。例如我们有个 Java 类，其中有个 is 方法：

```
public class MathTools {

    public boolean is(Object o) {
        return true;
    }

}
```

那么，我们在 Kotlin 代码这样调用这个 is 方法：

```
@RunWith(JUnit4::class)
class MathToolsTest {
    @Test fun testISKeyWord(){
        val b = MathTools().`is`(1)
    }
}
```

10.2.10　Nothing 类型

在 Kotlin 中 Nothing 类型是一个特殊的类型，它在 Java 中没有对应的类型。在使用 Nothing 参数的地方会生成一个原始类型。

例如下面的 Kotlin 代码：

```
fun emptyList(): List<Nothing> = listOf()
```

对应到 Java 代码中是：

```
@NotNull
public final List emptyList() {
    return CollectionsKt.emptyList();
}
```

Kotlin 中的 List<Nothing> 映射为原生类型 List。

10.3 类型映射

Kotlin 对一部分 Java 类型进行特殊处理，这样的类型不是“按原样”从 Java 加载，而是“映射“到相应的 Kotlin 类型。映射只发生在编译期间，运行时保持不变。

Java 的原生类型映射到相应的 Kotlin 类型如表 10-3 所示。

表 10-3 Java 的原生类型映射到相应的 Kotlin 类型

Java 类型	Kotlin 类型
byte	kotlin.Byte
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

Java 中的一些内置类型也会作相应的映射，参见表 10-4。

表 10-4 Java 的内置类型映射到相应的 Kotlin 类型

Java 类型	Kotlin 类型
java.lang.Object	kotlin.Any!
java.lang.Cloneable	kotlin.Cloneable!
java.lang.Comparable	kotlin.Comparable!
java.lang.Enum	kotlin.Enum!
java.lang.Annotation	kotlin.Annotation!
java.lang.Deprecated	kotlin.Deprecated!
java.lang.CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

Java 的装箱原始类型映射到对应的可空 Kotlin 类型，参见表 10-5。

表 10-5　Java 的装箱原始类型映射到对应的可空 Kotlin 类型

Java 类型	Kotlin 类型
java.lang.Byte	kotlin.Byte?
java.lang.Short	kotlin.Short?
java.lang.Integer	kotlin.Int?
java.lang.Long	kotlin.Long?
java.lang.Character	kotlin.Char?
java.lang.Float	kotlin.Float?
java.lang.Double	kotlin.Double?
java.lang.Boolean	kotlin.Boolean?

另外，用作类型参数的 Java 类型映射到 Kotlin 中的平台类型：例如，List<java.lang.Integer> 在 Kotlin 中会成为 List<Int!>。

集合类型在 Kotlin 中可以是只读的或可变的，因此 Java 集合类型的映射参见表 10-6。

表 10-6　Java 的集合类型映射到相应的 Kotlin 类型

Java 类型	Kotlin 只读类型	Kotlin 可变类型	加载的平台类型
Iterator<T>	Iterator<T>	MutableIterator<T>	(Mutable)Iterator<T>!
Iterable<T>	Iterable<T>	MutableIterable<T>	(Mutable)Iterable<T>!
Collection<T>	Collection<T>	MutableCollection<T>	(Mutable)Collection<T>!
Set<T>	Set<T>	MutableSet<T>	(Mutable)Set<T>!
List<T>	List<T>	MutableList<T>	(Mutable)List<T>!
ListIterator<T>	ListIterator<T>	MutableListIterator<T>	(Mutable)ListIterator<T>!
Map<K, V>	Map<K, V>	MutableMap<K, V>	(Mutable)Map<K, V>!
Map.Entry<K, V>	Map.Entry<K, V>	MutableMap.MutableEntry<K,V>	(Mutable)Map.(Mutable)Entry<K, V>!

注：表中的所有 Kotlin 类型都在 kotlin.collections 包中。

Java 的数组映射参见表 10-7。

表 10-7　Java 的数组类型映射到相应的 Kotlin 类型

Java 类型	Kotlin 类型
int[]	kotlin.IntArray!
String[]	kotlin.Array<(out) String>!

10.4　Kotlin 与 Java 的简单实例对比

在前面的内容里，我们已经看到了 Java 与 Kotlin 的互操作基本方式。为了更好地认识 Java 与 Kotlin 这两门语言，我们在这里同时使用 Java 与 Kotlin 来实现一些基本功能。通过

横向对比，我们可以直观地看出它们的异同。

1. 打印日志

Java	Kotlin
`System.out.print("Java");` `System.out.println("Java");`	`print("Kotlin")` `println("Kotlin")`

其实，Kotlin 中的 println 函数是一个内联函数，它是通过封装 java.lang.System 类的 System.out.println 来实现的：

```
@kotlin.internal.InlineOnly
public inline fun print(message: Any?) {
    System.out.print(message)
}
```

2. 常量与变量

Java	Kotlin
`String name = "KotlinVSJava";` `final String name = "KotlinVSJava";`	`var name = "KotlinVSJava"` `val name = "KotlinVSJava"`

3. null 声明

Java	Kotlin
`String otherName;` `otherName = null;`	`var otherName : String?` `otherName = null`

4. 空判断

Java	Kotlin
`if (text != null) {` `  int length = text.length();` `}`	`text?.let {` `  val length = text.length` `}` `// 或者` `val length = text?.length`

在 Kotlin 中，我们只使用一个问号安全调用符号就省去了 Java 中烦人的 if-null 判断。

5. 字符串拼接

Java	Kotlin
`String firstName = "Jack";` `String lastName = "Chen";` `String message = "My name is: " +` `  firstName + " " + lastName;`	`val firstName = "Jack"` `val lastName = "Chen"` `val message = "My name is: $firstName` `  $lastName"`

Kotlin 中使用 $ 和 ${}（花括号里面是表达式的时候）占位符来实现字符串的拼接，这个比在 Java 中每次使用加号来拼接要方便许多。

6. 换行

Java	Kotlin
`String text = "First Line\n" +` `"Second Line\n" +` `"Third Line";`	`val text = """` `|First Line` `|Second Line` `|Third Line` `""".trimMargin()`

7. 三元表达式

Java	Kotlin
`String text = x > 5 ? "x > 5" : "x <= 5";`	`val text = if (x > 5)` `"x > 5"` `else "x <= 5"`

8. 操作符

Java	Kotlin
`final int andResult  = a & b;` `final int orResult  = a | b;` `final int xorResult  = a ^ b;` `final int rightShift = a >> 2;` `final int leftShift  = a << 2;`	`val andResult  = a and b` `val orResult  = a or b` `val xorResult  = a xor b` `val rightShift = a shr 2` `val leftShift  = a shl 2`

9. 类型判断和转换（显式）

Java	Kotlin
`if (object instanceof Car) {` `}` `Car car = (Car) object;`	`if (object is Car) {` `}` `var car = object as Car`

10. 类型判断和转换（隐式）

Java	Kotlin
`if (object instanceof Car) {` `Car car = (Car) object;` `}`	`if (object is Car) {` `var car = object //Kotlin 智能转换` `}`

Kotlin 的类型系统具备一定的类型推断能力，这样也省去了不少在 Java 中类型转换的样板式代码。

11. 区间

Java	Kotlin
`if (score >= 0 && score <= 300) { }`	`if (score in 0..300) { }`

12. case 语句

Java

```
public String getGrade(int score) {
    String grade;
    switch (score) {
        case 10:
        case 9:
            grade = "A";
            break;
        case 8:
        case 7:
        case 6:
            grade = "B";
            break;
        case 5:
        case 4:
            grade = "C";
            break;
        case 3:
        case 2:
        case 1:
            grade = "D";
            break;
        default:
            grade = "E";
    }
    return grade;
}
```

Kotlin

```
fun getGrade(score: Int): String {
    var grade = when (score) {
        9, 10 -> "A"
        in 6..8 -> "B"
        4, 5 -> "C"
        in 1..3 -> "D"
        else -> "E"
    }
    return grade
}
```

13. for 循环

Java	Kotlin
`for (int i = 1; i <= 10 ; i++) { }`	`for (i in 1..10) { }`
`for (int i = 1; i < 10 ; i++) { }`	`for (i in 1 until 10) { }`
`for (int i = 10; i >= 0 ; i--) { }`	`for (i in 10 downTo 0) { }`
`for (int i = 1; i <= 10 ; i+=2) { }`	`for (i in 1..10 step 2) { }`
`for (int i = 10; i >= 0 ; i-=2) { }`	`for (i in 10 downTo 1 step 2) { }`
`for (String item : collection) { }`	`for (item in collection) { }`
`for (Map.Entry<String, String> entry: map.entrySet()) { }`	`for ((key, value) in map) { }`

14. 集合操作

Java	Kotlin
final List<Integer> listOfNumber = Arrays.asList(1, 2, 3, 4); final Map<Integer, String> map = new HashMap<Integer, String>(); map.put(1, "Jack"); map.put(2, "Ali"); map.put(3, "Mindorks");	val listOfNumber = listOf(1, 2, 3, 4) val map = mapOf(1 to "Jack", 2 to "Ali", 3 to "Mindorks")

15. 遍历

Java	Kotlin
// Java 7 for (Car car : cars) { System.out.println(car.speed); } // Java 8+ cars.forEach(car -> System.out.println(car.speed)); // Java 7 for (Car car : cars) { if (car.speed > 100) { System.out.println(car.speed); } } // Java 8+ cars.stream().filter(car -> car.speed > 100).forEach(car -> System.out.println(car.speed));	// Kotlin cars.forEach { println(it.speed) } cars.filter { it.speed > 100 } .forEach { println(it.speed)}

16. 方法 (函数) 定义

Java	Kotlin
void doSomething() { // 实现 } void doSomething(int... numbers) { // 实现 }	fun doSomething() { // 实现 } fun doSomething(vararg numbers: Int) { // 实现 }

17. 带返回值的方法（函数）

Java	Kotlin
int getScore() { // logic here return score; }	fun getScore(): Int { // logic here return score } // 单表达式函数 fun getScore(): Int = score

另外，Kotlin 中的函数是可以直接传入函数参数，同时可以返回一个函数类型的。

18. 构造器

Java	Kotlin
public class Utils { private Utils() { // 外部无法来调用实例化 } public static int getScore(int value) { return 2 * value; } }	class Utils private constructor() { companion object { fun getScore(value: Int): Int { return 2 * value } } } // 或者直接声明一个 object 对象 object Utils { fun getScore(value: Int): Int { return 2 * value } }

19. JavaBean 与 Kotlin 数据类

Kotlin 中的数据类代码如下：

```
data class Developer(val name: String, val age: Int)
```

对应下面这段为 Java 实体类的代码：

```
public final class Developer {
    @NotNull
    private final String name;
    private final int age;

    @NotNull
```

```
public final String getName() {
    return this.name;
}

public final int getAge() {
    return this.age;
}

public Developer(@NotNull String name, int age) {
    Intrinsics.checkParameterIsNotNull(name, "name");
    super();
    this.name = name;
    this.age = age;
}

@NotNull
public final String component1() {
    return this.name;
}

public final int component2() {
    return this.age;
}

@NotNull
public final Developer copy(@NotNull String name, int age) {
    Intrinsics.checkParameterIsNotNull(name, "name");
    return new Developer(name, age);
}

// $FF: synthetic method
// $FF: bridge method
@NotNull
public static Developer copy$default(Developer var0, String var1, int var2,
    int var3, Object var4) {
    if((var3 & 1) != 0) {
        var1 = var0.name;
    }

    if((var3 & 2) != 0) {
        var2 = var0.age;
    }

    return var0.copy(var1, var2);
}

public String toString() {
    return "Developer(name=" + this.name + ", age=" + this.age + ")";
}

public int hashCode() {
```

```
        return (this.name != null?this.name.hashCode():0) * 31 + this.age;
    }

    public boolean equals(Object var1) {
        if(this != var1) {
            if(var1 instanceof Developer) {
                Developer var2 = (Developer)var1;
                if(Intrinsics.areEqual(this.name, var2.name) && this.age ==
                    var2.age) {
                    return true;
                }
            }

            return false;
        } else {
            return true;
        }
    }
}
```

10.5 本章小结

本章我们一起学习了 Kotlin 与 Java 的互操作，同时用一些简单的示例对比了它们的异同。在这之中，我们能感受到 Kotlin 的简洁、优雅，我们可以用更少的代码来实现更多的功能。另外，在 IDEA 中，可以直接使用 Kotlin 插件进行 Java 代码与 Kotlin 代码之间的转换（虽然，有些情况下需要手工稍作改动）。

Kotlin 的定位之一就是官网首页重点强调的：100% interoperable with Java。它并不像 Scala 那样另起炉灶，将类库（例如集合类）都自己实现了一遍。Kotlin 是对现有 Java 的增强，通过扩展方法给 Java 提供了很多诸如 FP 之类的特性，但同时始终保持对 Java 的兼容。

Spring 框架是 Java 生态领域最为常用的框架，其最新的 Spring 5 版本已支持 Kotlin（参见：https://spring.io/blog/2017/01/04/introducing-kotlin-support-in-spring-framework-5-0）。当前，作为 Spring 大家族中最引人注目的非 Spring Boot 莫属了。我们将在下一章中介绍如何用 Kotlin 集成 Spring Boot 开发 Web 服务端项目。

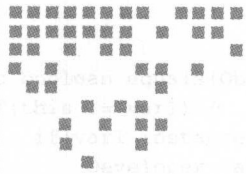
本章示例代码：https://github.com/EasyKotlin/chapter10_interoperability

---

第三部分 Part 3

综合实战

- 第 11 章　使用 Kotlin 集成 Spring Boot 开发 Web 服务端
- 第 12 章　使用 Kotlin 集成 Gradle 开发
- 第 13 章　使用 Kotlin 和 Anko 的 Android 开发
- 第 14 章　使用 Kotlin DSL
- 第 15 章　Kotlin 文件 IO 操作与多线程
- 第 16 章　使用 Kotlin Native



Chapter 11 第 11 章

使用 Kotlin 集成 Spring Boot 开发 Web 服务端

我们在前面第 2 章 2.3 节“ Web RESTFul HelloWorld”中，已经介绍了使用 Kotlin 结合 Spring Boot 开发一个 RESTFul 版本的 Hello World。当然，Kotlin 与 Spring 家族的关系不止如此。在 Spring 5.0 M4 中引入了一个专门针对 Kotlin 的支持。

本章我们就一起来学习怎样使用 Kotlin 集成 Spring Boot、Spring MVC 等框架来开发 Web 服务端应用，同时简单介绍 Spring 5.0 对 Kotlin 的支持特性。

11.1 Spring Boot 简介

Spring 由于其繁琐的配置，一度被人认为“配置地狱”，各种 XML 文件的配置，让人眼花缭乱，而且如果出错了也很难找出原因。Spring Boot 正是为了解决这些问题而诞生。

Spring Boot 是伴随着 Spring 4.0 诞生的。从字面理解，Boot 是引导的意思，Spring Boot 帮助开发者快速搭建 Spring 框架、快速启动一个 Web 容器等，使得基于 Spring 的开发过程更加简易。大部分 Spring Boot 应用只要一些极简的配置即可“一键运行”。

Spring Boot 的特性如下：

- 创建独立的 Spring applications。
- 能够使用内嵌的 Tomcat、Jetty 或 Undertow，不需要部署 War。
- 提供定制化的 starter poms 来简化 Maven 配置（Gradle 相同）。
- 追求极致的自动配置 Spring。
- 提供一些生产环境的特性，比如特征指标、健康检查和外部配置。
- 零代码生成和零 XML 配置。

11.2　系统架构技术栈

本节介绍使用 Kotlin 集成 Spring Boot 开发一个完整的博客站点的服务端 Web 应用，支持 Markdown 写文章、文章列表分页、搜索查询等功能，其系统架构技术栈如下所示：

- 编程语言：Java、Kotlin
- 数据库：MySQL、mysql-jdbc-driver、Spring data JPA
- J2EE 框架：Spring Boot、Spring MVC
- 视图模板引擎：Freemarker
- 前端组件库：Jquery、Bootstrap、flat UI、Mditor、DataTables
- 工程构建工具：Gradle

11.3　准备工作

11.3.1　创建工程

首先，使用 Spring Initializr 来创建一个模板项目。

1）访问 http://start.spring.io/，选择生成一个 Gradle 项目，使用 Kotlin 语言，使用的 Spring Boot 版本是 2.0.0 M2。

2）配置项目基本信息：

```
Group: com.easy.kotlin
Artifact: chapter11_kotlin_springboot
```

以及项目名称、项目描述、包名称等其他的选项。选择 jar 包方式打包，使用 JDK1.8。

3）选择项目依赖。我们这里分别选择了：Web、DevTools、JPA、MySQL、Actuator、Freemarker。

以上三步如图 11-1 所示。

点击“生成项目”，下载 zip 包，解压后导入 IDEA 中，我们可以看到一个如下目录结构的项目：

```
.
├── build
│   └── kotlin-build
│       └── caches
├── build.gradle
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
└── gradlew.bat
```

```
└── src
    ├── main
    │   ├── kotlin
    │   │   └── com
    │   │       └── easy
    │   │           └── kotlin
    │   │               └── chapter11_kotlin_springboot
    │   │                   └── Chapter11KotlinSpringbootApplication.kt
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── kotlin
            └── com
                └── easy
                    └── kotlin
                        └── chapter11_kotlin_springboot
                            └── Chapter11KotlinSpringbootApplicationTests.kt

21 directories, 8 files
```

其中，Chapter11KotlinSpringbootApplication.kt 是 Spring Boot 应用的入口启动类。

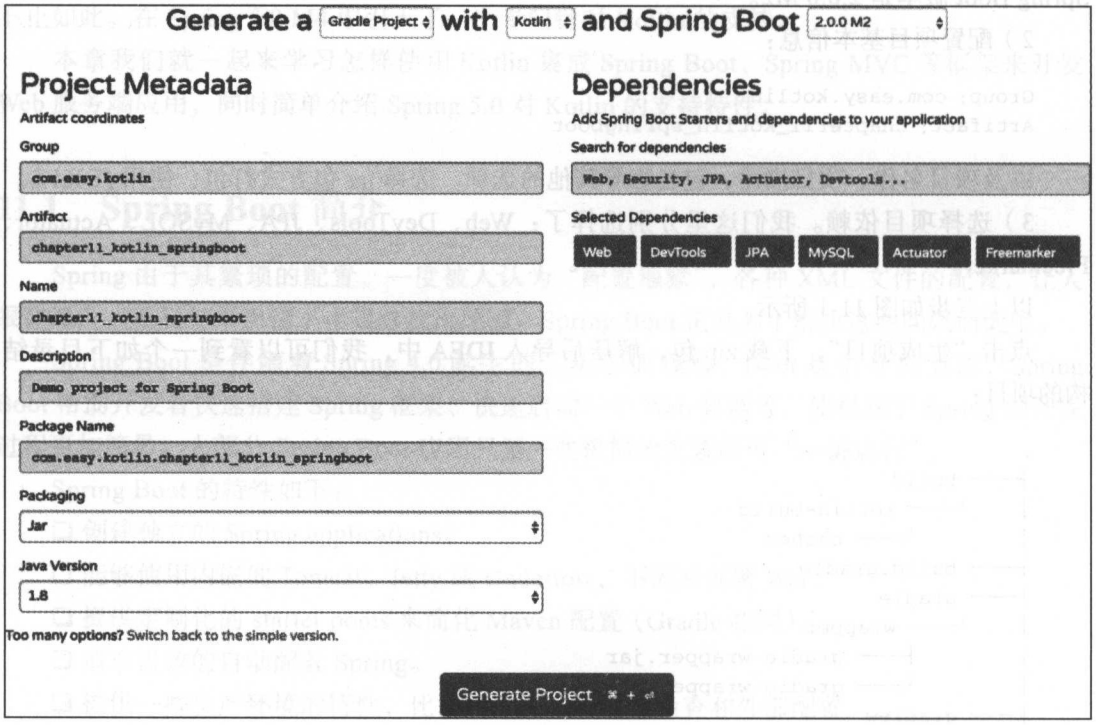


图 11-1 创建项目

11.3.2　Gradle 配置文件说明

整个项目的 Gradle 构建配置文件 build.gradle 的内容如下：

```
buildscript {
    ext {
        kotlinVersion = '1.1.3-2'
        springBootVersion = '2.0.0.M2'
    }
    repositories {
        mavenCentral()
        maven { url "https://repo.spring.io/snapshot" }
        maven { url "https://repo.spring.io/milestone" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:
            ${springBootVersion}")
        classpath("org.jetbrains.kotlin:kotlin-gradle-plugin:${kotlinVersion}")
        classpath("org.jetbrains.kotlin:kotlin-allopen:${kotlinVersion}")
    }
}

apply plugin: 'kotlin'
apply plugin: 'kotlin-spring'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8
compileKotlin {
    kotlinOptions.jvmTarget = "1.8"
}
compileTestKotlin {
    kotlinOptions.jvmTarget = "1.8"
}

repositories {
    mavenCentral()
    maven { url "https://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/milestone" }
}


dependencies {
    compile('org.springframework.boot:spring-boot-starter-actuator')
    compile('org.springframework.boot:spring-boot-starter-data-jpa')
    compile('org.springframework.boot:spring-boot-starter-freemarker')
    compile('org.springframework.boot:spring-boot-starter-web')
    compile("org.jetbrains.kotlin:kotlin-stdlib-jre8:${kotlinVersion}")
    compile("org.jetbrains.kotlin:kotlin-reflect:${kotlinVersion}")
```

```
    runtime('org.springframework.boot:spring-boot-devtools')
    runtime('mysql:mysql-connector-java')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

其中主要的配置项如表 11-1 所示。

表 11-1 主要的配置项说明

配置项	功能说明
spring-boot-gradle-plugin	Spring Boot 集成 Gradle 的插件
kotlin-gradle-plugin	Kotlin 集成 Gradle 的插件
kotlin-allopen	Kotlin 全开放插件。Kotlin 里类默认都是 final 的，如果声明的类需要被继承则需要使用 open 关键字来描述类，这个插件就是把 Kotlin 中的所有类都 open 打开，可被继承
spring-boot-starter-actuator	Spring Boot 的健康检查监控组件启动器
spring-boot-starter-data-jpa	JPA 启动器
spring-boot-starter-freemarker	模板引擎 freemarker 启动器
kotlin-stdlib-jre8	Kotlin 基于 JRE8 的标准库
kotlin-reflect	Kotlin 反射库
spring-boot-devtools	Spring Boot 开发者工具，例如：热部署等
mysql-connector-java	Java 的 MySQL 连接器库
spring-boot-starter-test	测试启动器

11.4 数据库层开发

上面的模板项目中，我们直接运行 main 函数，会发现启动失败，控制台会输出如下报错信息：

```
BeanCreationException: Error creating bean with name 'dataSource' defined
    in class path resource [org/springframework/boot/autoconfigure/jdbc/
    DataSourceConfiguration$Hikari.class]
...
***************************
APPLICATION FAILED TO START
***************************

Description:

Cannot determine embedded database driver class for database type NONE

Action:

If you want an embedded database please put a supported one on the classpath.
    If you have database settings to be loaded from a particular profile you
    may need to active it (no profiles are currently active).
```

因为我们还没有配置数据源。我们先在 MySQL 中新建一个 schema：

```
CREATE SCHEMA `blog` DEFAULT CHARACTER SET utf8 ;
```

接着，在配置文件 application.properties 中配置 MySQL 数据源：

```
# datasource
# datasource: unicode 编码的支持，设定为 utf-8
spring.datasource.url=jdbc:mysql://localhost:3306/blog?zeroDateTimeBehavior=
    convertToNull&characterEncoding=utf8&characterSetResults=utf8
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.testWhileIdle=true
spring.datasource.validationQuery=SELECT 1
```

其中，spring.datasource.url 是为了支持中文的正确显示（防止出现中文乱码），我们需要设置一下编码。

对数据库 ORM（对象关系映射）层，我们使用 spring-data-jpa：

```
spring.jpa.database=MYSQL
spring.jpa.show-sql=true
# Hibernate ddl auto (create, create-drop, update)
spring.jpa.hibernate.ddl-auto=update
# Naming strategy
spring.jpa.hibernate.naming-strategy=org.hibernate.cfg.ImprovedNamingStrategy
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

再次运行启动类，控制台输出启动日志：

```
...
...

2017-07-17 21:10:48.741  INFO 5062 --- [  restartedMain]
    c.Chapter11KotlinSpringbootApplicationKt : Starting
    Chapter11KotlinSpringbootApplicationKt on 192.168.1.6 with PID 5062 ...
...
2017-07-17 21:19:40.548  INFO 5329 --- [  restartedMain]
    j.LocalContainerEntityManagerFactoryBean : Initialized JPA
    EntityManagerFactory for persistence unit 'default'
2017-07-17 21:11:03.017  INFO 5062 --- [  restartedMain]
    o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/application/env ||
    /application/env.json],methods=[GET],produces=[application/vnd.spring-
    boot.actuator.v2+json || application/json]}" onto public java.lang.Object
    org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invoke()
...
2017-07-17 21:11:03.026  INFO 5062 --- [  restartedMain]
    o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/application/beans ||
    /application/beans.json],methods=[GET],produces=[application/vnd.spring-
    boot.actuator.v2+json || application/json]}" onto public java.lang.Object
    org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invoke()
```

```
2017-07-17 21:11:03.028  INFO 5062 --- [  restartedMain]
    o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/application/health ||
    /application/health.json],methods=[GET],produces=[application/vnd.spring-
    boot.actuator.v2+json || application/json]}" onto public java.lang.Object
    org.springframework.boot.actuate.endpoint.mvc.HealthMvcEndpoint.invoke
    (javax.servlet.http.HttpServletRequest,java.security.Principal)
...
2017-07-17 21:11:03.060  INFO 5062 --- [  restartedMain]
    o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/application/
    autoconfig || /application/autoconfig.json],methods=[GET],produces=
    [application/vnd.spring-boot.actuator.v2+json || application/json]}"
    onto public java.lang.Object org.springframework.boot.actuate.endpoint.
    mvc.EndpointMvcAdapter.invoke()
...
2017-07-17 21:11:03.478  INFO 5062 --- [  restartedMain]
    o.s.ui.freemarker.SpringTemplateLoader   : SpringTemplateLoader for
    FreeMarker: using resource loader [org.springframework.boot.web.servlet.
    context.AnnotationConfigServletWebServerApplicationContext@6aab8872:
    startup date [Mon Jul 17 21:10:48 CST 2017]; root of context hierarchy]
    and template loader path [classpath:/templates/]
2017-07-17 21:11:03.479  INFO 5062 --- [  restartedMain]
    o.s.w.s.v.f.FreeMarkerConfigurer         : ClassTemplateLoader for Spring
    macros added to FreeMarker configuration
2017-07-17 21:11:03.520  WARN 5062 --- [  restartedMain]
    o.s.b.a.f.FreeMarkerAutoConfiguration    : Cannot find template location(s):
    [classpath:/templates/] (please add some templates, check your FreeMarker
    configuration, or set spring.freemarker.checkTemplateLocation=false)
2017-07-17 21:11:03.713  INFO 5062 --- [  restartedMain]
    o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running
    on port 35729
2017-07-17 21:11:03.871  INFO 5062 --- [  restartedMain]
    o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX
    exposure on startup
2017-07-17 21:11:03.874  INFO 5062 --- [  restartedMain]
    o.s.j.e.a.AnnotationMBeanExporter        : Bean with name 'dataSource'
    has been autodetected for JMX exposure
2017-07-17 21:11:03.886  INFO 5062 --- [  restartedMain]
    o.s.j.e.a.AnnotationMBeanExporter        : Located MBean 'dataSource':
    registering with JMX server as MBean [com.zaxxer.hikari:name=dataSource,
    type=HikariDataSource]
2017-07-17 21:11:03.901  INFO 5062 --- [  restartedMain]
    o.s.c.support.DefaultLifecycleProcessor  : Starting beans in phase 0
2017-07-17 21:11:04.232  INFO 5062 --- [  restartedMain]
    o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s):
    8000 (http)
2017-07-17 21:11:04.240  INFO 5062 --- [  restartedMain]
    c.Chapter11KotlinSpringbootApplicationKt : Started
    Chapter11KotlinSpringbootApplicationKt in 16.316 seconds
    (JVM running for 17.68)
```

关于上面的日志，我们用表 11-2 简要说明。

表 11-2　日志内容说明

日志内容	简要说明
`LocalContainerEntityManagerFactoryBean: Initialized JPA EntityManagerFactory`	初始化 JAP 实体管理器工厂
`EndpointHandlerMapping: Mapped "{[/application/beans …等`	Spring Boot 健康监控 Endpoint 等 REST 接口
`FreeMarkerAutoConfiguration`	Freemarker 模板引擎自动配置，默认视图文件目录是 classpath:/templates/
`AnnotationMBeanExporter: Bean with name'dataSource' has been autodetected for JMX exposure … Located MBean 'dataSource': registering with JMX server as MBean [com.zaxxer.hikari:name=dataSource,type=HikariDataSource]`	数据源 Bean 通过 annotation 注解注册 MBean 到 JMX 实现监控其运行状态
`TomcatWebServer: Tomcat started on port(s): 8000 (http)`	Spring Boot 默认内嵌了 Tomcat，可以在 application.properties 中配置端口
`Started Chapter11KotlinSpringbootApplicationKt in 16.316 seconds (JVM running for 17.68)`	Spring Boot 应用启动成功

11.4.1　Endpoint 监控接口

我们来尝试访问：http://127.0.0.1:8000/application/beans，浏览器显示如下信息：

```
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as
    a fallback.

Mon Jul 17 21:30:35 CST 2017
There was an unexpected error (type=Unauthorized, status=401).
Full authentication is required to access this resource.
```

提示没有权限访问。我们去控制台日志可以看到下面这行输出：

```
s.b.a.e.m.MvcEndpointSecurityInterceptor : Full authentication is required
    to access actuator endpoints. Consider adding Spring Security or set
    'management.security.enabled' to false.
```

意思是说，要访问这些 Endpoints 需要权限，可以通过 Spring Security 来实现权限控制，或者把权限限制去掉：把 management.security.enabled 设置为 false。

我们在 application.properties 里面添加配置：

```
management.security.enabled=false
```

重启应用，再次访问，我们可以看到如下输出：

```
[
    {
        "context": "application:8000",
        "parent": null,
```

```
    "beans": [
        {
            "bean": "chapter11KotlinSpringbootApplication",
            "aliases": [

            ],
            "scope": "singleton",
            "type": "com.easy.kotlin.chapter11_kotlin_springboot.Chapter11-
                KotlinSpringbootApplication$$EnhancerBySpringCGLIB$$353fd63e",
            "resource": "null",
            "dependencies": [

            ]
        },
        ...
        {
            "bean": "org.springframework.boot.autoconfigure.orm.jpa.
                HibernateJpaAutoConfiguration",
            "aliases": [

            ],
            "scope": "singleton",
            "type": "org.springframework.boot.autoconfigure.orm.jpa.
                HibernateJpaAutoConfiguration$$EnhancerBySpringCGLIB$$24d31c25",
            "resource": "null",
            "dependencies": [
                "dataSource",
                "spring.jpa-org.springframework.boot.autoconfigure.orm.
                    jpa.JpaProperties"
            ]
        },
        {
            "bean": "transactionManager",
            "aliases": [

            ],
            "scope": "singleton",
            "type": "org.springframework.orm.jpa.JpaTransactionManager",
            "resource": "class path resource [org/springframework/boot/
                autoconfigure/orm/jpa/HibernateJpaAutoConfiguration.class]",
            "dependencies": [

            ]
        },
        ...
        {
            "bean": "spring.devtools-org.springframework.boot.devtools.
                autoconfigure.DevToolsProperties",
            "aliases": [

            ],
```

```
            "scope": "singleton",
            "type": "org.springframework.boot.devtools.autoconfigure.
                DevToolsProperties",
            "resource": "null",
            "dependencies": [

            ]
        },
        {
            "bean": "org.springframework.orm.jpa.SharedEntityManagerCreator#0",
            "aliases": [

            ],
            "scope": "singleton",
            "type": "com.sun.proxy.$Proxy86",
            "resource": "null",
            "dependencies": [
                "entityManagerFactory"
            ]
        }
    ]
}
```

可以看出，我们一行代码还没写，只是加了几行配置，Spring Boot 已经自动配置初始化了这么多的 Bean。我们再访问 http://127.0.0.1:8000/application/health：

```
{
    "status": "UP",
    "diskSpace": {
        "status": "UP",
        "total": 120108089344,
        "free": 1724157952,
        "threshold": 10485760
    },
    "db": {
        "status": "UP",
        "database": "MySQL",
        "hello": 1
    }
}
```

从上面可以看到一些应用的健康状态信息，例如：应用状态、磁盘空间、数据库状态等信息。

11.4.2　实体类

我们在上面已经完成了 MySQL 数据源的配置，下面来写一个实体类。新建 package com.easy.kotlin.chapter11_kotlin_springboot.entity，然后新建 Article 实体类：

```
package com.easy.kotlin.chapter11_kotlin_springboot.entity

import java.util.*
import javax.persistence.*

@Entity
class Article {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var id: Long = -1
    @Version
    var version: Long = 0
    var title: String = ""
    var content: String = ""
    var author: String = ""
    var gmtCreated: Date = Date()
    var gmtModified: Date = Date()
    var isDeleted: Int = 0 //1 Yes 0 No
    var deletedDate: Date = Date()

    override fun toString(): String {
        return "Article(id=$id, version=$version, title='$title', content=
            '$content', author='$author', gmtCreated=$gmtCreated, gmtModified=
            $gmtModified, isDeleted=$isDeleted, deletedDate=$deletedDate)"
    }

}
```

类似的实体类，我们在 Java 中需要生成一堆 Getter/Setter 方法；如果我们用 Scala 写还需要加个注解 @BeanProperty，例如：

```
package com.springboot.in.action.entity

import java.util.Date
import javax.persistence.{ Entity, GeneratedValue, GenerationType, Id }
import scala.language.implicitConversions
import scala.beans.BeanProperty

@Entity
class HttpApi {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @BeanProperty
    var id: Integer = _

    @BeanProperty
    var httpSuiteId: Integer = _
    //用例名称
    @BeanProperty
    var name: String = _

    //用例状态：-1 未执行 0 失败 1 成功
```

```
    @BeanProperty
    var state: Integer = _
    ...

}
```

我们的目标是建立博客文章的简单实体类。再次重启运行应用，我们去 MySQL 的 Schema: blog 里面去看，发现数据库自动生成了 Table: article，它的表字段信息如表 11-3 所示。

表 11-3　数据库 Table: article 的表字段信息

Field	Type	Null	Key	Default	Extra
id	bigint(20)	NO	PRI	NULL	auto_increment
author	varchar(255)	YES		NULL	
content	varchar(255)	YES		NULL	
deleted_date	datetime	YES		NULL	
gmt_created	datetime	YES		NULL	
gmt_modified	datetime	YES		NULL	
is_deleted	int(11)	NO		NULL	
title	varchar(255)	YES		NULL	
version	bigint(20)	NO		NULL	-

11.4.3　数据访问层代码

在 Spring Data JPA 中，我们只需要实现接口 CrudRepository<T, ID> 即可获得一个拥有基本 CRUD 操作的接口实现了：

```
interface ArticleRepository : CrudRepository<Article, Long>
```

JPA 会自动实现 ArticleRepository 接口中的方法，不需要我们写基本的 CRUD 操作代码，常用的基本 CRUD 操作方法参见表 11-4。

表 11-4　常用的基本 CRUD 操作方法

方　法	功能说明
S save(S entity)	保存给定的实体对象，可以使用这个保存之后返回的实例进行进一步操作（保存操作可能会更改实体实例）
findById(ID id)	根据主键 id 查询
existsById(ID id)	判断是否存在该主键 id 的记录
findAll()	返回所有记录
findAllById(Iterable ids)	根据主键 id 集合批量查询
count()	返回总记录条数
deleteById(ID id)	根据主键 id 删除
deleteAll()	全部删除

当然，如果我们需要自己去实现 SQL 查询逻辑，可以直接使用 @Query 注解：

```
interface ArticleRepository : CrudRepository<Article, Long> {
    override fun findAll(): MutableList<Article>

    @Query(value = "SELECT * FROM blog.article where title like %?1%",
        nativeQuery = true)
    fun findByTitle(title: String): MutableList<Article>

    @Query("SELECT a FROM #{#entityName} a where a.content like %:content%")
    fun findByContent(@Param("content") content: String): MutableList<Article>

    @Query(value = "SELECT * FROM blog.article  where author = ?1",
        nativeQuery = true)
    fun findByAuthor(author: String): MutableList<Article>
}
```

下面简单介绍一下其中的技巧。

1. 原生 SQL 查询

其中，@Query 注解里面的 value 值就是我们要写的 JP QL 语句。另外，JPA 的 EntityManager API 还提供了创建 Query 实例以执行原生 SQL 语句的 createNativeQuery 方法。

默认是非原生的 JP QL 查询模式。如果我们想指定原生 SQL 查询，只需要设置 nativeQuery=true 即可。

2. 模糊查询 like 写法

另外，原生 SQL 模糊查询 like 语法在写 SQL 的时候是这样写的：

```
like '%?%'
```

但是在 JP QL 中这样写：

```
like %?1%
```

3. 参数占位符

查询语句中的 ?1 是函数参数的占位符，1 代表的是参数的位置。

4. JP QL 中的 SpEL

我们使用 JPA 的标准查询（Criteria Query）：

```
SELECT a FROM #{#entityName} a where a.content like %:content%
```

其中的 #{#entityName} 是 SpEL（Spring 表达式语言），用来代替本来实体的名称，而 Spring data jpa 会自动根据 Article 实体上对应的默认的 @Entity class Article，或者指定 @Entity(name = "Article") class Article 自动将实体名称填入 JP QL 语句中。

通过把实体类名称抽象出来成为参数，简化了 SQL 的编写。

5. 注解参数

我们使用 @Param("content") 来指定参数名绑定，然后在 JP QL 语句中这样引用：

```
:content
```

JP QL 语句中通过“:变量”的格式来指定参数，同时在方法的参数前面使用 @Param 将方法参数与 JP QL 中的命名参数对应。

11.5　后端逻辑开发

本节介绍后端逻辑的开发，包括控制器层、应用启动类、初始化逻辑、视图层模板引擎等。

11.5.1　控制器逻辑层

我们新建子目录 controller，然后在下面新建控制器类：

```
@Controller
class ArticleController {

}
```

首先装配数据访问层的接口 Bean：

```
@Autowired val articleRepository: ArticleRepository? = null
```

这个接口 Bean 的实例化由 Spring data jpa 完成。如果我们去 http://127.0.0.1:8000/application/beans 中查看这个 Bean，可以看到信息如下：

```
{
    "bean": "articleRepository",
    "aliases": [

    ],
    "scope": "singleton",
    "type": "com.easy.kotlin.chapter11_kotlin_springboot.dao.ArticleRepository",
    "resource": "null",
    "dependencies": [
        "(inner bean)#39c36d98",
        "(inner bean)#19d60142",
        "(inner bean)#1757cb01",
        "(inner bean)#6dd045f0",
        "jpaMappingContext"
    ]
}
```

我们先来实现一个简单的查询所有记录的 REST 接口。我们在 ArticleRepository 中重写

了 findAll 方法：

```
override fun findAll(): MutableList<Article>
```

然后，我们在控制器代码中直接调用这个接口方法：

```
@GetMapping("listAllArticle")
@ResponseBody
fun listAllArticle(): MutableList<Article>? {
    return articleRepository?.findAll()
}
```

其中，注解 @ResponseBody 表示把方法返回值直接绑定到响应体（response body）。

11.5.2 启动初始化 CommandLineRunner

为了方便测试用，我们在 Spring Boot 应用启动的时候初始化几条数据到数据库里。Spring Boot 为我们提供了一个方法，通过实现接口 CommandLineRunner 来实现。这是一个函数式接口：

```
@FunctionalInterface
public interface CommandLineRunner {
    void run(String... args) throws Exception;
}
```

我们只需要创建一个实现接口 CommandLineRunner 的类。真的很简单，只需要一个类就可以，无需其他配置。这里我们使用 Kotlin 的 Lambda 表达式来写：

```
@Bean
fun init(repository: ArticleRepository) = CommandLineRunner {
    val article: Article = Article()
    article.author = "Kotlin"
    article.title = "极简 Kotlin 教程 ${Date()}"
    article.content = "Easy Kotlin ${Date()}"
    repository.save(article)
}
```

11.5.3 应用启动类

我们在 main 函数中调用 SpringApplication 类的静态 run 方法，我们的 SpringBootApplication 主类代码如下：

```
package com.easy.kotlin.chapter11_kotlin_springboot

import com.easy.kotlin.chapter11_kotlin_springboot.dao.ArticleRepository
import com.easy.kotlin.chapter11_kotlin_springboot.entity.Article
import org.springframework.boot.CommandLineRunner
import org.springframework.boot.SpringApplication
import org.springframework.boot.autoconfigure.SpringBootApplication
```

```
import org.springframework.context.annotation.Bean
import java.util.*

@SpringBootApplication
class Chapter11KotlinSpringbootApplication {
    @Bean
    fun init(repository: ArticleRepository) = CommandLineRunner {
        val article: Article = Article()
        article.author = "Kotlin"
        article.title = "极简 Kotlin 教程 ${Date()}"
        article.content = "Easy Kotlin ${Date()}"
        repository.save(article)
    }
}

fun main(args: Array<String>) {
    SpringApplication.run(Chapter11KotlinSpringbootApplication::class.java, *args)
}
```

这里主要关注的是 @SpringBootApplication 注解，它包括三个注解，用表 11-5 简单说明。

表 11-5　@SpringBootApplication 注解说明

注　解	功能说明
@SpringBootConfiguration（包括 @Configuration）	表示将该类作用于 Spring Boot 配置文件类
@EnableAutoConfiguration	表示 Spring Boot 程序启动时，启动 Spring Boot 默认的自动配置
@ComponentScan	表示程序启动时自动扫描当前包及子包下所有类

如果是在 IDEA 中运行，可以直接点击 main 函数运行，如图 11-2 所示。

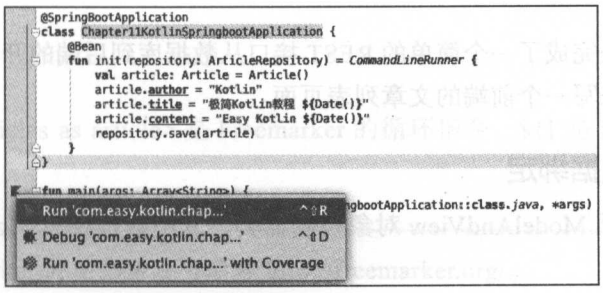


图 11-2　直接点击 main 函数运行

如果想在命令行运行，直接在项目根目录下运行命令：

```
$ gradle bootRun
```

我们可以看到控制台的日志输出：

```
2017-07-18 17:42:53.689  INFO 21239 --- [  restartedMain] o.s.b.w.embedded.
    tomcat.TomcatWebServer  : Tomcat started on port(s): 8000 (http)
Hibernate: insert into article (author, content, deleted_date, gmt_created,
    gmt_modified, is_deleted, title, version) values (?, ?, ?, ?, ?, ?, ?, ?)
2017-07-18 17:42:53.974  INFO 21239 --- [  restartedMain] c.Chapter11Kotlin-
    SpringbootApplicationKt : Started Chapter11KotlinSpringbootApplicationKt
    in 16.183 seconds (JVM running for 17.663)
<-------------> 83% EXECUTING [1m 43s]
> :bootRun
```

我们在浏览器中直接访问：http://127.0.0.1:8000/listAllArticle，可以看到类似如下输出：

```
[
    {
        "id": 1,
        "version": 0,
        "title": "极简Kotlin教程",
        "content": "Easy Kotlin ",
        "author": "Kotlin",
        "gmtCreated": 1500306475000,
        "gmtModified": 1500306475000,
        "deletedDate": 1500306475000
    },
    {
        "id": 2,
        "version": 0,
        "title": "极简Kotlin教程",
        "content": "Easy Kotlin ",
        "author": "Kotlin",
        "gmtCreated": 1500306764000,
        "gmtModified": 1500306764000,
        "deletedDate": 1500306764000
    }
]
```

至此，我们已经完成了一个简单的 REST 接口从数据库到后端的开发。

下面我们继续来写一个前端的文章列表页面。

11.5.4 Model 数据绑定

我们写一个返回 ModelAndView 对象控制器类，其中数据模型 Model 中放入文章列表数据，代码如下：

```
@GetMapping("listAllArticleView")
fun listAllArticleView(model: Model): ModelAndView {
    model.addAttribute("articles", articleRepository?.findAll())
    return ModelAndView("list")
}
```

其中，ModelAndView("list") 中的“ list”表示视图文件所在目录的相对路径。Spring

Boot 的默认视图文件放在 src/main/resources/templates 目录下。

11.5.5　模板引擎视图页面

我们使用 Freemarker 模板引擎。在 templates 目录下新建一个 list.ftl 文件，内容如下：

```
<html>
<head>
    <title>Blog!!!</title>
</head>
<body>
<table>
    <thead>
    <th> 序号 </th>
    <th> 标题 </th>
    <th> 作者 </th>
    <th> 发表时间 </th>
    <th> 操作 </th>
    </thead>
    <tbody>
    <#-- 使用 FTL 指令 -->
    <#list articles as article>
    <tr>
        <td>${article.id}</td>
        <td>${article.title}</td>
        <td>${article.author}</td>
        <td>${article.gmtModified}</td>
        <td><a href="#" target="_blank"> 编辑 </a></td>
    </tr>
    </#list>
    </tbody>
</table>
</body>
</html>
```

其中，<#list articles as article> 是 Freemarker 的循环指令，${} 是 Freemarker 引用变量的方式。



关于 Freemarker 的详细语法可参考 http://freemarker.org/。

11.5.6　运行测试

重启应用，浏览器访问：http://127.0.0.1:8000/listAllArticleView，我们可以看到页面输出，如图 11-3 所示。

序号	标题	作者	发表时间	操作
1	极简Kotlin教程	Kotlin	2017-7-17 23:47:55	编辑
2	极简Kotlin教程	Kotlin	2017-7-17 23:52:44	编辑
3	极简Kotlin教程	Kotlin	2017-7-17 23:53:11	编辑
4	极简Kotlin教程	Kotlin	2017-7-17 23:54:46	编辑
5	极简Kotlin教程	Kotlin	2017-7-17 23:57:49	编辑
6	极简Kotlin教程	Kotlin	2017-7-18 0:04:37	编辑
7	极简Kotlin教程	Kotlin	2017-7-18 0:07:20	编辑
8	极简Kotlin教程 Tue Jul 18 00:11:05 CST 2017	Kotlin	2017-7-18 0:11:06	编辑
9	极简Kotlin教程 Tue Jul 18 00:16:46 CST 2017	Kotlin	2017-7-18 0:16:46	编辑
10	极简Kotlin教程 Tue Jul 18 00:20:05 CST 2017	Kotlin	2017-7-18 0:20:06	编辑
11	极简Kotlin教程 Tue Jul 18 09:58:27 CST 2017	Kotlin	2017-7-18 9:58:27	编辑
12	极简Kotlin教程 Tue Jul 18 10:07:09 CST 2017	Kotlin	2017-7-18 10:07:10	编辑
13	极简Kotlin教程 Tue Jul 18 10:31:31 CST 2017	Kotlin	2017-7-18 10:31:31	编辑
14	极简Kotlin教程 Tue Jul 18 10:50:21 CST 2017	Kotlin	2017-7-18 10:50:22	编辑
15	极简Kotlin教程 Tue Jul 18 11:00:50 CST 2017	Kotlin	2017-7-18 11:00:50	编辑
16	极简Kotlin教程 Tue Jul 18 15:44:54 CST 2017	Kotlin	2017-7-18 15:44:54	编辑
17	极简Kotlin教程 Tue Jul 18 17:42:53 CST 2017	Kotlin	2017-7-18 17:42:54	编辑
18	极简Kotlin教程 Tue Jul 18 17:49:57 CST 2017	Kotlin	2017-7-18 17:49:58	编辑
19	极简Kotlin教程 Tue Jul 18 23:27:51 CST 2017	Kotlin	2017-7-18 23:27:52	编辑

图 11-3 运行测试

到这里，我们已经完成了一个从数据库到前端页面的极简的 Web 应用。

当然，这样的 UI 样式未免太简陋了一些。下面我们加入前端 UI 组件美化一下。

11.6 前端后端集成

为了美化页面，我们引入了前端组件。本节介绍写文章、文章详情模块的前端代码，以及前后端集成的相关步骤。

11.6.1 引入前端组件

我们使用基于 Bootstrap 的前端 UI 库 Flat UI。首先去 Flat UI 的首页：http://www.bootcss.com/p/flat-ui/ 下载 zip 包，加压后，放到我们的项目里，放置的目录是：src/main/resources/static。如图 11-4 所示。

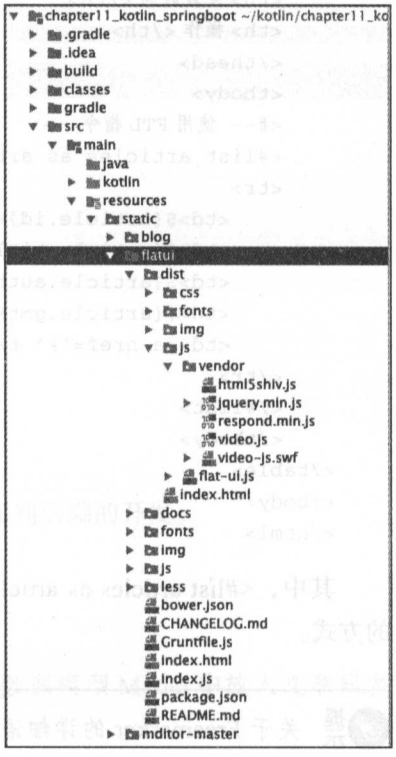


图 11-4 引入前端组件

我们在 list.ftl 头部引入静态资源文件：

```
<head>
    <meta charset="utf-8">
    <title>Blog</title>
    <meta name="description"
          content="Blog, using Flat UI Kit Free is a Twitter Bootstrap Framework
              design and Theme, this responsive framework includes a PSD and
              HTML version."/>
```

```
    <meta name="viewport" content="width=1000, initial-scale=1.0, maximum-
        scale=1.0">

    <!-- Loading Bootstrap -->
    <link href="/flatui/dist/css/vendor/bootstrap/css/bootstrap.min.css" rel=
        "stylesheet">

    <!-- Loading Flat UI -->
    <link href="/flatui/dist/css/flat-ui.css" rel="stylesheet">
    <link href="/flatui/docs/assets/css/demo.css" rel="stylesheet">

    <link rel="shortcut icon" href="/flatui/img/favicon.ico">

    <script src="/flatui/dist/js/vendor/jquery.min.js"></script>
    <script src="/flatui/dist/js/flat-ui.js"></script>
    <script src="/flatui/dist/js/vendor/html5shiv.js"></script>
    <script src="/flatui/dist/js/vendor/respond.min.js"></script>

    <link rel="stylesheet" href="/blog/blog.css">
    <script src="/blog/blog.js"></script>
</head>
```

其中，我们的 Spring Boot 应用中默认的静态资源的跟路径是 src/main/resources/static，然后我们的 HTML 代码中引用的路径是在此根目录下的相对路径。

提示　更多的关于 Spring Boot 静态资源处理内容可以参考文章：http://www.jianshu.com/p/d127c4f78bb8

然后，我们再把我们的文章列表布局优化一下：

```
<div class="container">
    <h1>我的博客</h1>
    <table class="table table-responsive table-bordered">
        <thead>
        <th>序号</th>
        <th>标题</th>
        <th>作者</th>
        <th>发表时间</th>
        <th>操作</th>
        </thead>
        <tbody>
        <#-- 使用 FTL 指令 -->
        <#list articles as article>
        <tr>
            <td>${article.id}</td>
            <td>${article.title}</td>
            <td>${article.author}</td>
            <td>${article.gmtModified}</td>
```

```
            <td><a href="#" target="_blank">编辑</a></td>
        </tr>
        </#list>
        </tbody>
    </table>
</div>
```

重新构建项目，再次访问文章列表页，我们将看到一个比刚才漂亮多了的页面，如图 11-5 所示。

我的博客

序号	标题	作者	发表时间	操作
1	极简Kotlin教程	Kotlin	2017-7-17 23:47:55	编辑
2	极简Kotlin教程	Kotlin	2017-7-17 23:52:44	编辑
3	极简Kotlin教程	Kotlin	2017-7-17 23:53:11	编辑
4	极简Kotlin教程	Kotlin	2017-7-17 23:54:46	编辑
5	极简Kotlin教程	Kotlin	2017-7-17 23:57:49	编辑
6	极简Kotlin教程	Kotlin	2017-7-18 0:04:37	编辑
7	极简Kotlin教程	Kotlin	2017-7-18 0:07:20	编辑
8	极简Kotlin教程 Tue Jul 18 00:11:05 CST 2017	Kotlin	2017-7-18 0:11:06	编辑
9	极简Kotlin教程 Tue Jul 18 00:16:46 CST 2017	Kotlin	2017-7-18 0:16:46	编辑
10	极简Kotlin教程 Tue Jul 18 00:20:05 CST 2017	Kotlin	2017-7-18 0:20:06	编辑
11	极简Kotlin教程 Tue Jul 18 09:58:27 CST 2017	Kotlin	2017-7-18 9:58:27	编辑
12	极简Kotlin教程 Tue Jul 18 10:07:09 CST 2017	Kotlin	2017-7-18 10:07:10	编辑
13	极简Kotlin教程 Tue Jul 18 10:31:31 CST 2017	Kotlin	2017-7-18 10:31:31	编辑
14	极简Kotlin教程 Tue Jul 18 10:50:21 CST 2017	Kotlin	2017-7-18 10:50:22	编辑

图 11-5 重新构建工程

考虑到头部的静态资源文件基本都是公共的代码，我们单独抽取到一个 head.ftl 文件，然后在 list.ftl 中直接这样引用：

```
<#include "head.ftl">
```

11.6.2 实现写文章模块

我们在列表页上面添加一个“写文章”的入口：

```
<a href="addArticleView" target="_blank" class="btn btn-primary pull-right
    add-article">写文章</a>
```

其中，btn、btn-primary、pull-right 这三个 CSS 样式类是 Flat UI 组件的。add-article 是我们自定义的样式类：

```
.add-article {
    margin: 20px;
}
```

下面我们来写新建文章的页面。写文章的跳转页面路径是 <a href="addArticleView">，我们先来新建一个写文章页面 addArticleView.ftl：

```
<!DOCTYPE html>
<html>
<#include "head.ftl">
<body>
<div class="container">
    <h2>写文章 </h2>

    <form id="addArticleForm" class="form-horizontal">
        <div class="form-group">
            <input type="text" name="title" class="form-control" placeholder=
                " 文章标题 ">
        </div>

        <div class="form-group">
            <textarea id="articleContentEditor" type="text" name="content"
                class="form-control" rows="20"
                    placeholder=""></textarea>
        </div>

        <div class="form-group save-article">
            <div class="col-sm-offset-2 col-sm-10">
                <button type="submit" class="btn btn-primary" id="addArticleBtn">
                    保存并发表 </button>
            </div>
        </div>
    </form>
</div>
</body>
</html>
```

然后，再添加控制器请求转发。这里使用集成 WebMvcConfigurerAdapter 类，重写实现 addViewControllers 方法的方式来添加一个不带数据传输的、单纯的请求转发的跳转 View 的 RequestMapping Controller：

```
@Configuration
class WebMvcConfig : WebMvcConfigurerAdapter() {
    //注册简单请求转发跳转 View 的 RequestMapping Controller
    override fun addViewControllers(registry: ViewControllerRegistry?) {
        //写文章的 RequestMapping
        registry?.addViewController("addArticleView")?.setViewName("addArticleView")
    }
}
```

这样前端浏览器来的请求 addArticle 会直接映射转发到视图 addArticle.ftl 文件渲染解析。重启应用，进入到我们的写文章的页面，如图 11-6 所示。

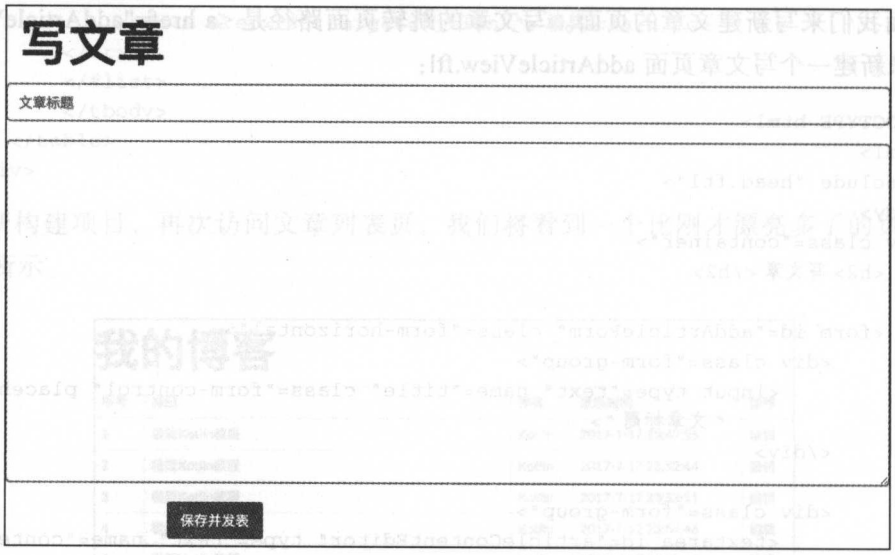


图 11-6 写文章的页面

11.6.3 加上导航栏

为了方便页面之间的跳转，我们给博客站点加上导航栏。新建一个 navbar.ftl 文件，内容如下：

```
<nav class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle" data-toggle="collapse"
                    data-target="#example-navbar-collapse">
                <span class="sr-only">切换导航</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">我的博客</a>
        </div>
        <div class="collapse navbar-collapse" id="example-navbar-collapse">
            <ul class="nav navbar-nav">
                <li class=""><a href="listAllArticleView">文章列表</a></li>
                <li class="active"><a href="addArticleView">写文章</a></li>
                <li><a href="#">关于</a></li>
                <li class="dropdown">
                    <a href="http://www.jianshu.com/nb/12976878" class=
                        "dropdown-toggle" data-toggle="dropdown">
                        Kotlin <b class="caret"></b>
                    </a>
                    <ul class="dropdown-menu">
                        <li><a href="http://www.jianshu.com/nb/12976878" target=
```

```
                    "_blank">Kotlin 极简教程 </a></li>
                <li class="divider"></li>
                <li><a href="#">Java</a></li>
                <li><a href="#">Scala</a></li>
                <li><a href="#">Groovy</a></li>
                <li class="divider"></li>
                <li><a href="http://www.jianshu.com/nb/12066555" target=
                    "_blank">SpringBoot 极简教程 </a></li>
            </ul>
        </li>
    </ul>
  </div>
 </div>
</nav>
```

11.6.4　抽取公共模板文件

考虑到 head.ftl、navbar.ftl 都是公共的文件，我们把它们单独放到一个 common 目录下。然后，我们分别在 addArticleView.ftl、listAllArticleView.ftl 中使用 <#include> 标签引用 head.ftl、navbar.ftl，代码如下：

```
<!DOCTYPE html>
<html>
<#include "common/head.ftl">
<body>
<#include "common/navbar.ftl">
```

加入了导航栏之后，我们的页面现在更加美观了，如图 11-7 和图 11-8 所示。



图 11-7　加上导航栏（一）

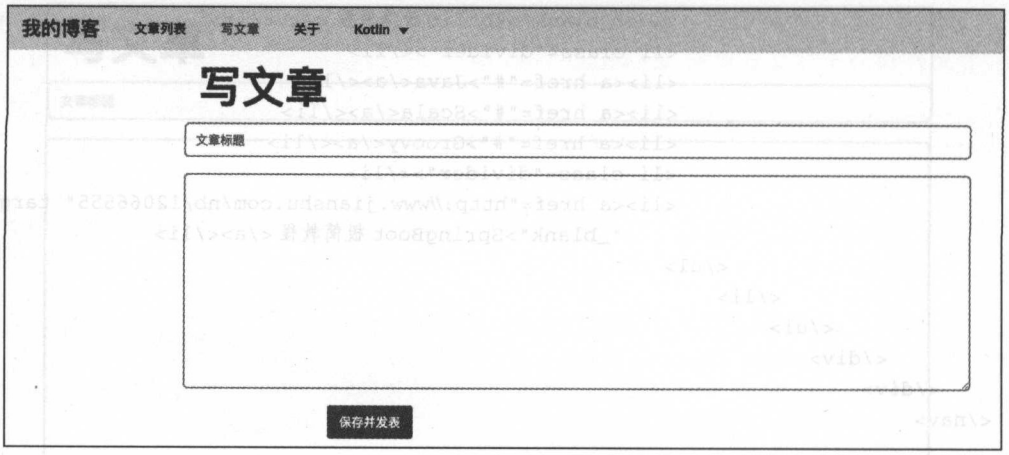


图 11-8 加上导航栏（二）

11.6.5 写文章的控制器层接口

控制器层保存接口代码如下：

```
@PostMapping("saveArticle")
@ResponseBody
fun saveArticle(article: Article): Article? {
    article.gmtCreated = Date()
    article.gmtModified = Date()
    article.version = 0
    return articleRepository?.save(article)
}
```

另外，为了支持较多内容的文章，我们把文章内容字段设置成 LONGTEXT：

```
ALTER TABLE `blog`.`article`
CHANGE COLUMN `content` `content` LONGTEXT NULL DEFAULT NULL ;
```

11.6.6 前端 Ajax 请求

上面完成了控制器层的 http 接口，现在来完成前端的调用逻辑。我们在 blog.js 中加上 Ajax POST 请求后端接口的逻辑如下所示：

```
$(function () {

    $('#addArticleBtn').on('click', function () {
        saveArticle()
    })

    function saveArticle() {
        $.ajax({
            url: "saveArticle",
```

```
            data: $('#addArticleForm').serialize(),
            type: "POST",
            async: false,
            success: function (resp) {
                if (resp) {
                    saveArticleSuccess(resp)
                } else {
                    saveArticleFail()
                }
            },
            error: function () {
                saveArticleFail()
            }
        })
    }

    function saveArticleSuccess(resp) {
        alert('保存成功：' + JSON.stringify(resp))
        window.open('detailArticleView?id=' + resp.id)
    }

    function saveArticleFail() {
        alert("保存失败！")
    }


})
```

11.6.7　文章详情页

保存成功后，我们默认跳转该文章详情页。

后端控制器代码如下：

```
@GetMapping("detailArticleView")
fun detailArticleView(id: Long, model: Model): ModelAndView {
    model.addAttribute("article", articleRepository?.findById(id)?.get())
    return ModelAndView("detailArticleView")
}
```

注意，这里的 articleRepository?.findById(id) 方法返回的是 Optional，我们调用其 get() 方法，返回真正的 Article 实体对象。

前端视图 detailArticleView.ftl 代码如下：

```
<!DOCTYPE html>
<html>
<#include "common/head.ftl">
<body>
<#include "common/navbar.ftl">
<div class="container">
    <h3>${article.title}</h3>
```

```
    <h6>${article.author}</h6>
    <h6>${article.gmtModified}</h6>
    <div>${article.content}</div>
</div>
</body>
</html>
```

我们在文章列表页中，给每篇文章标题加上跳转文章详情的超链接：

```
<td><a target="_blank" href="detailArticleView?id=${article.id}">${article.
    title}</a></td>
```

现在我们的文章列表页面如图 11-9 所示。



图 11-9 文章列表页面

点击一篇文章标题，即可进入详情页，如图 11-10 所示。

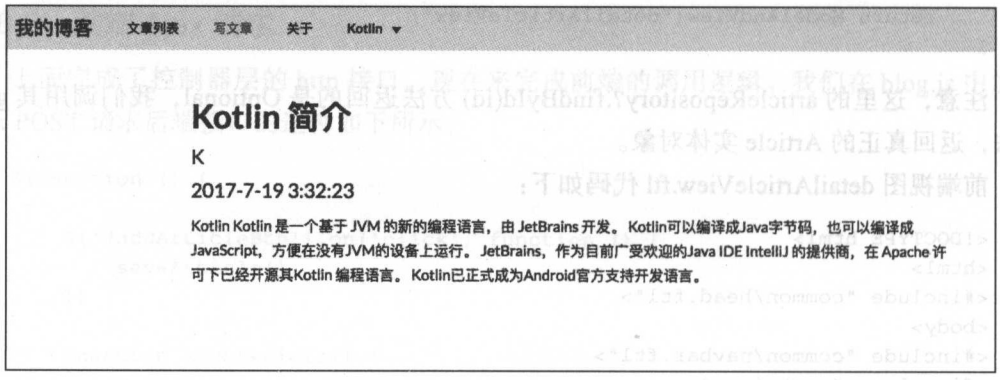


图 11-10 文章详情页面

11.6.8　添加 Markdown 支持

我们写技术博客文章最常用的就是 Markdown 了，我们来为博客添加 Markdown 的支持。我们使用前端 js 组件 Mditor 来支持 Markdown 的编辑，Mditor 是一个简洁、易于集成、方便扩展、期望舒服的编写 markdown 的编辑器。具体步骤如下。

1）引入静态资源：

```
<link href="/mditor-master/dist/css/mditor.css" rel="stylesheet">
<script src="/mditor-master/dist/js/mditor.js"></script>
```

2）初始化 Mditor。我们在写文章的页面 addArticleView.ftl，初始化 Mditor 如下：

```
<script>
    $(function () {
        //写文章 mditor
        var mditor = Mditor.fromTextarea(document.getElementById
            ('articleContentEditor'));

        //是否打开分屏
        mditor.split = true;                    //打开
        //是否打开预览
        mditor.preivew = true;                  //打开
        //是否全屏
        mditor.fullscreen = false;              //关闭
        //获取或设置编辑器的值
        mditor.on('ready', function () {
            mditor.value = '# ';
        });
        hljs.initHighlightingOnLoad();
        //源码高亮
        $('pre code').each(function (i, block) {
            hljs.highlightBlock(block);
        });
    })
</script>
```

另外，我们还使用了代码高亮插件 highlight.js。

这样，写文章的页面对应的 textarea 区域就变成了支持 Markdown 在线编辑 + 预览的功能了，如图 11-11 所示。

3）文章详情显示 Markdown 渲染。下面我们来使详情页也能支持 Markdown 的渲染显示。详情页的视图文件 detailArticleView.ftl 如下：

```
<!DOCTYPE html>
<html>
<#include "common/head.ftl">
<body>
<#include "common/navbar.ftl">
<div class="container">
```

```
    <h3>${article.title}</h3>
    <h6>${article.author}</h6>
    <h6>${article.gmtModified}</h6>
    <textarea id="articleContentShow" placeholder="<#escape  x as x?html>
        ${article.content}</#escape>" style="display:
            none"></textarea>
    <div id="article-content" class="markdown-body"></div>
</div>
</body>
</html>
```

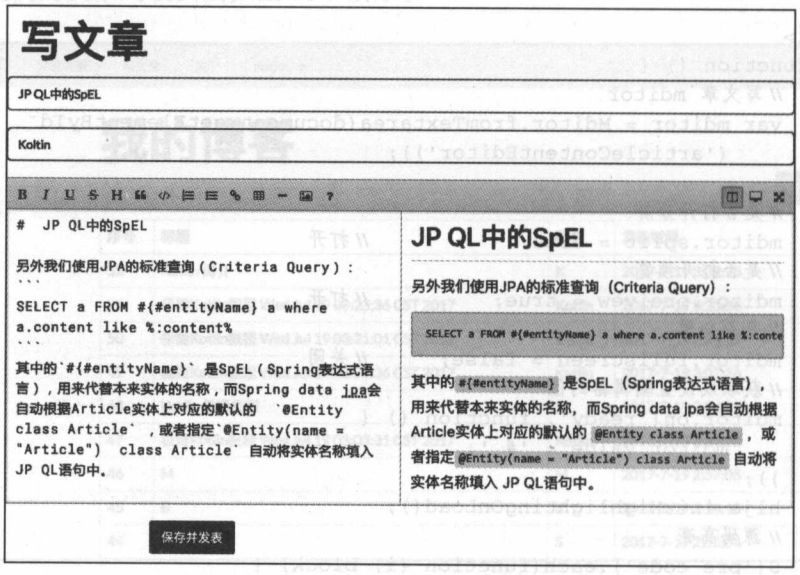


图 11-11 支持 Markdown 在线编辑 + 预览

这里我们把文章内容放到一个隐藏的 textarea 的 placeholder 属性中：

```
<textarea id="articleContentShow" placeholder="<#escape  x as x?html>${article.
    content}</#escape>" style="display:
        none"></textarea>
```

注意，这里我们作了字符的转义 escape，防止有特殊字符导致页面显示错乱。

然后，我们在 js 中获取这个内容：

```
<script>
    $(function () {
        // 文章详情 mditor
        var parser = new Mditor.Parser();
        var articleContent = document.getElementById('articleContentShow').
            placeholder          // 直接取原本的字符串。不会被转译，默认 html 页面中
                                 // textarea 区域 text 需要 escape 编码
```

```
        articleContent = unescape(articleContent);//unescape 解码
        var html = parser.parse(articleContent);

        $('#article-content').append(html);

        hljs.initHighlightingOnLoad();
        //源码高亮
        $('pre code').each(function (i, block) {
            hljs.highlightBlock(block);
        });
    })
</script>
```

其中，我们是直接调用 Mditor.Parser() 函数来解析 Markdown 字符文本的。这样，我们的详情页也支持了 Markdown 的渲染显示了，如图 11-12 所示。

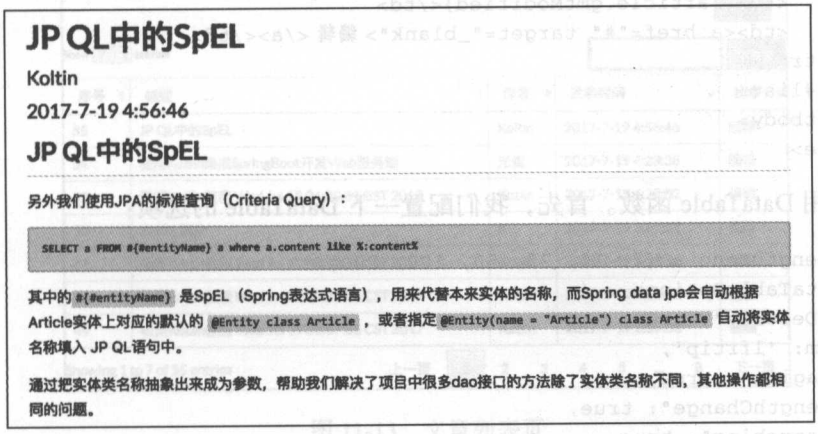


图 11-12　支持 Markdown 的渲染显示

11.6.9　文章列表分页搜索

为了方便检索博客文章，我们再来给文章列表页面添加分页、搜索、排序等功能。我们使用前端组件 DataTables 来实现。具体步骤如下。


提示　更多关于 DataTables，可参考：http://www.datatables.club/

1）引入静态资源文件：

```
<link href="/datatables/media/css/jquery.dataTables.css" rel="stylesheet">
<script src="/datatables/media/js/jquery.dataTables.js"></script>
```

2）给表格加上 id。我们给表格加个属性 id=“articlesDataTable”：

```
<table id="articlesDataTable" class="table table-responsive table-bordered">
    <thead>
    <th>序号</th>
    <th>标题</th>
    <th>作者</th>
    <th>发表时间</th>
    <th>操作</th>
    </thead>
    <tbody>
    <#-- 使用 FTL 指令 -->
    <#list articles as article>
    <tr>
        <td>${article.id}</td>
        <td><a target="_blank" href="detailArticleView?id=${article.id}">
            ${article.title}</a></td>
        <td>${article.author}</td>
        <td>${article.gmtModified}</td>
        <td><a href="#" target="_blank">编辑</a></td>
    </tr>
    </#list>
    </tbody>
</table>
```

3）调用 DataTable 函数。首先，我们配置一下 DataTable 的选项：

```
var aLengthMenu = [7, 10, 20, 50, 100, 200]
var dataTableOptions = {
    "bDestroy": true,
    dom: 'lfrtip',
    "paging": true,
    "lengthChange": true,
    "searching": true,
    "ordering": true,
    "info": true,
    "autoWidth": true,
    "processing": true,
    "stateSave": true,
    responsive: true,
    fixedHeader: false,
    order: [[1, "desc"]],
    "aLengthMenu": aLengthMenu,
    language: {
        "search": "<div style='border-radius:10px;margin-left:auto;margin-
            right:2px;width:760px;'>_INPUT_  <span class='btn btn-primary'>
            <span class='fa fa-search'></span> 搜索</span></div>",

        paginate: {//分页的样式内容
            previous: "上一页",
            next: "下一页",
            first: "第一页",
            last: "最后"
```

```
        }
    },
    zeroRecords: "没有内容",//table tbody 内容为空时，tbody 的内容
    //下面三者构成了总体的左下角的内容
    info: "总计 _TOTAL_ 条，共 _PAGES_ 页，_START_ - _END_ ",
    //左下角的信息显示，大写的词为关键字
    infoEmpty: "0 条记录",//筛选为空时左下角的显示
    infoFiltered: ""//筛选之后的左下角筛选提示
}
```

然后把我们刚才添加了 id 的表格使用 JQuery 选择器获取对象，再直接调用：

```
$('#articlesDataTable').DataTable(dataTableOptions)
```

再次看我们的文章列表页，已经具备了分页、搜索、排序等功能了。如图 11-13 所示。

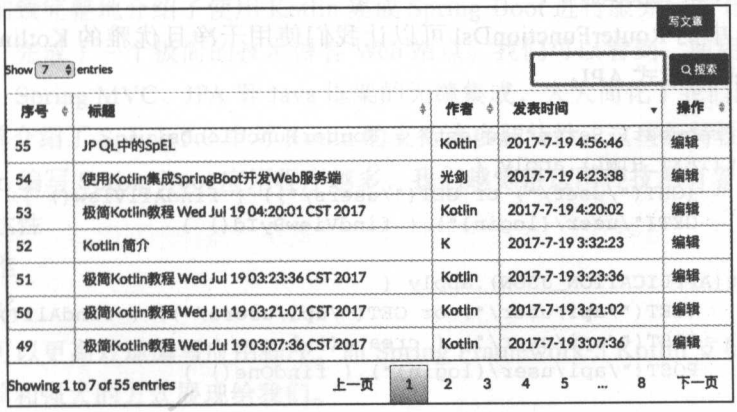


图 11-13　文章列表页

到这里，这个较为完整的极简博客站点应用基本就开发完成了。

11.7　Spring 5.0 对 Kotlin 的支持

Kotlin 关键性能之一就是能与 Java 库很好地互用。但要在 Spring 中编写惯用的 Kotlin 代码，还需要一段时间的发展。Spring 对 Java 8 的新支持：函数式 Web 编程、bean 注册 API，这同样可以在 Kotlin 中使用。

Kotlin 扩展是 Kotlin 的编程利器，它能对现有的 API 实现非侵入式的扩展，从而向 Spring 中加入 Kotlin 的专有功能特性。

1. 一种注册 Bean 的新方法

Spring Framework 5.0 引入了一种注册 Bean 的新方法，作为利用 XML 或者 JavaConfig 的 @Configuration 或者 @Bean 的替代方案，简言之就是 Lambda 表达式。

例如用 Java 代码我们会这样写：

```
GenericApplicationContext context = new GenericApplicationContext();
context.registerBean(Foo.class);
context.registerBean(Bar.class, () -> new
    Bar(context.getBean(Foo.class))
);
```

而使用 Kotlin 我们可以将代码写成这样：

```
val context = GenericApplicationContext {
    registerBean<foo>()
    registerBean { Bar(it.getBean<foo>()) }
}
```

2. Spring Web 函数式 API

Spring 5.0 中的 RouterFunctionDsl 可以让我们使用干净且优雅的 Kotlin 代码来使用崭新的 Spring Web 函数式 API：

```
fun route(request: ServerRequest) = RouterFunctionDsl {
    accept(TEXT_HTML).apply {
            (GET("/user/") or GET("/users/")) { findAllView() }
            GET("/user/{login}") { findViewById() }
    }
    accept(APPLICATION_JSON).apply {
            (GET("/api/user/") or GET("/api/users/")) { findAll() }
            POST("/api/user/") { create() }
            POST("/api/user/{login}") { findOne() }
    }
} (request)
```

3. Reactor Kotlin 扩展

Reactor 是 Spring 5.0 中提供的响应式框架。而 reactor-kotlin 项目则是对 Reactor 中使用 Kotlin 的支持。目前该项目正在早期阶段。

4. 基于 Kotlin 脚本的 Gradle 构建配置

之前我们的 Gradle 构建配置文件都是用 Groovy 来编写的，这导致我们基于 Gradle 的 Kotlin 项目还要配置 Groovy 语法的构建配置文件。

在 gradle-script-kotlin 项目中，我们可以直接用 Kotlin 脚本来编写 Gradle 的构建配置文件了，而且 IDE 还为我们提供了在编写配置文件过程中的自动完成功能和重构功能。

5. 基于模板的 Kotlin 脚本

从 4.3 版本开始，Spring 提供了一个 ScriptTemplateView，方便使用支持 JSR-223 的脚本引擎来渲染模板。Kotlin 1.1-M04 提供了这样的支持，并支持渲染基于 Kotlin 的模板，类似下面这样：

```
import io.spring.demo.User
import io.spring.demo.joinToLine

"""
${include("header", bindings)}
<h1>Title : $title</h1>
<ul>
    ${(users as List<User>).joinToLine{ "<li>User ${it.firstname} ${it.lastname}</li>" }}
</ul>
${include("footer")}
"""
```

11.8　本章小结

本章较为细致完整地介绍了使用 Kotlin 集成 Spring Boot 进行服务后端开发，并结合简单的前端开发，完成了一个极简的技术博客 Web 站点。我们可以看到，使用 Kotlin 可以结合 Spring Boot、Spring MVC、JPA 等 Java 框架的无缝集成，大大简化了我们的代码。同时，在本章最后简单介绍了 Spring 5.0 中对 Kotlin 的支持诸多新特性，这些新特性令人惊喜。

使用 Kotlin 编写 Spring Boot 应用程序越多，我们越觉得这两种技术有着共同的目标：

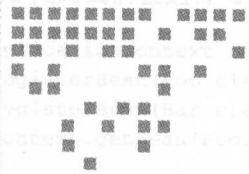
- 富有表达性
- 简洁优雅
- 可读性好

使程序员可以更高效地编写应用程序，而 Spring Framework 5 Kotlin 支持将这些技术以更加自然、简单和强大的方式展现给我们。

未来 Spring Framework 5.0 和 Kotlin 结合的开发实践更加值得我们期待。

在下一章中我们将一起学习 Kotlin 集成 Gradle 开发的相关内容。

本章示例代码：https://github.com/EasyKotlin/chapter11_kotlin_springboot



Chapter 12 第 12 章

使用 Kotlin 集成 Gradle 开发

由于 Kotlin 具有丰富的功能，如一等函数和扩展方法等，因此它可以保留和改进 Gradle 构建脚本的最佳部分——包括简明的声明式语法以及轻松制作 DSL 的能力。

Gradle 团队与 Kotlin 团队密切合作，为 Gradle 开发了新的基于 Kotlin 脚本的构建配置语言，我们称之为 Gradle Script Kotlin，支持使用 Kotlin 编写构建和配置文件，同时，还支持在 IDE 中实现自动完成和编译检查等功能。有了 Gradle Script Kotlin，我们可以使用 Kotlin 来写配置文件，就像写普通代码一样。

我们在前面的章节中，已经有很多示例项目使用了 Gradle 来构建 Kotlin 项目。本章将系统地介绍一下使用 Kotlin 集成 Gradle 开发的相关内容。

12.1 使用 Gradle 构建 Kotlin 工程

Gradle 是一个项目自动化构建工具，基于 Ant 和 Maven，它使用基于 Groovy DSL（领域特定语言）来声明项目设置，抛弃了基于 XML 的各种繁琐配置。

我们使用 kotlin-gradle 插件来构建 Kotlin 语言的 Gradle 项目。

1. kotlin-gradle 插件

为了用 Gradle 构建 Kotlin 工程，我们需要设置好 kotlin-gradle 插件。首先我们添加构建依赖项：

```
buildscript {
    ext {
        kotlinVersion = '1.1.3-2'
```

```
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.jetbrains.kotlin:kotlin-gradle-plugin:${kotlinVersion}")
                ...
    }
}
```

然后，应用 kotlin 插件：

```
apply plugin: 'kotlin'
```

并且添加 kotlin-stdlib 依赖：

```
dependencies {
    compile("org.jetbrains.kotlin:kotlin-stdlib-jre8:${kotlinVersion}")
    compile("org.jetbrains.kotlin:kotlin-reflect:${kotlinVersion}")
}
```

当然，这些操作在我们新建项目的时候，通常只需选择相应的选项，IntelliJ IDEA 就会直接帮我们完成基本的配置。

我们使用 kotlin-gradle 插件编译 Kotlin 源代码和模块。使用的 Kotlin 版本通常定义为 kotlinVersion 属性。

针对 JVM，需要应用 Kotlin 插件 apply plugin: "kotlin"。

2. Kotlin 与 Java 混合编程

Gradle 的资源路径一般都是固定的，例如：

```
src/main/java/***
src/test/java/***
```

也就是说我们再不编辑 sourceSets 的情况下，只有这里面的代码才会被编辑，test 中的是测试代码。

Kotlin 源代码可以与同一个文件夹或不同文件夹中的 Java 源代码混用。默认约定是使用不同的文件夹：

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/kotlin'
    main.java.srcDirs += 'src/main/java'
}
```

如果使用默认的目录，上面的配置可以省去不写。

如果不使用默认约定，那么应该更新相应的 sourceSets 属性：

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
```

```
    main.java.srcDirs += 'src/main/myJava'
}
```

这样新添加的源代码文件目录会被编译处理。

3. 配置 Gradle JavaScript 项目

当针对 JavaScript 时，须应用不同的插件：

```
apply plugin: "kotlin2js"
```

除了输出的 JavaScript 文件，该插件默认会创建一个带二进制描述符的额外 JS 文件。

如果是构建其他 Kotlin 模块可以依赖的可重用库，那么该文件是必需的，并且与转换结果一起分发。

二进制描述符文件的生成由 kotlinOptions.metaInfo 选项控制：

```
compileKotlin2Js {
    kotlinOptions.metaInfo = true
}
```



示例项目可以参考 https://github.com/EasyKotlin/chapter2_hello_world_kotlin2js

4. 配置 Gradle Android 项目

Android 的 Gradle 模型与普通 Gradle 有点不同，所以如果我们要构建一个用 Kotlin 编写的 Android 项目，需要用 kotlin-android 插件取代 kotlin 插件：

```
buildscript {
    ext.kotlin_version = '1.1.2-4'
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

通常，我们使用 Android Studio，都是生成一个带 app 子项目的工程。多项目配置的实现通常是在一个根项目路径下将所有项目作为子文件夹包含进去。例如，我们在项目根路径下面的 settings.gradle 中如下配置：

```
include ':app'
```

每一个子项目都拥有自己的 build.gradle 文件来声明自己如何构建。

例如，我们在子项目 app 的构建配置文件 build.gradle 中一个完整的配置如下：

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "com.kotlin.easy.kotlinandroid"
        minSdkVersion 14
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.3.1'
    compile 'com.android.support.constraint:constraint-layout:1.0.2'
    testCompile 'junit:junit:4.12'
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version"
}
repositories {
    mavenCentral()
}
```

其中：

- apply plugin: ‘kotlin-android’ 是 Kotlin Android 插件。
- compile "org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version" 是 Kotlin 运行标准库。

另外，Android Studio 默认加载源码的目录是 src/main/java，如果想指定 Kotlin 代码在 src/main/kotln 目录，可以在 android 下添加以下内容：

```
android {
    ......
    sourceSets {
        main.java.srcDirs += 'src/main/kotlin'
    }
}
```


提示 关于 Kotlin Android 的 Gradle 完整配置实例可参考 https://github.com/EasyKotlin/KotlinAndroid。

5. 配置 Kotlin 标准库依赖

除了上面的 kotlin-gradle-plugin 依赖之外，还需要添加 Kotlin 标准库的依赖：

```
repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib"
}
```

如果针对 JavaScript，使用 compile "org.jetbrains.kotlin:kotlin-stdlib-js" 替代之。

如果是针对 JDK 7 或 JDK 8，那么可以使用扩展版本的 Kotlin 标准库，其中包含为新版 JDK 增加的额外的扩展函数。使用以下依赖之一来取代 kotlin-stdlib：

```
compile "org.jetbrains.kotlin:kotlin-stdlib-jre7"
compile "org.jetbrains.kotlin:kotlin-stdlib-jre8"
```

如果项目中使用 Kotlin 反射，添加反射依赖：

```
compile "org.jetbrains.kotlin:kotlin-reflect"
```

如果项目中使用测试框架，则需添加相应的测试库依赖：

```
testCompile "org.jetbrains.kotlin:kotlin-test"
testCompile "org.jetbrains.kotlin:kotlin-test-junit"
```

6. 增量编译

Kotlin 支持 Gradle 中可选的增量编译。增量编译跟踪构建之间源文件的改动，因此只有受这些改动影响的文件才会被编译。从 Kotlin 1.1.1 起，默认启用增量编译。

7. 编译器选项

要指定附加的编译选项，可以使用 Kotlin 编译任务 compileKotlin 的 kotlinOptions 属性。配置单个任务示例：

```
compileKotlin {
    kotlinOptions {
        suppressWarnings = true
    }
}
```


提示 更多关于使用 Gradle 构建 Kotlin 工程的配置可参考官方文档：http://kotlinlang.org/docs/reference/using-gradle.html

12.2　使用 Kotlin 编写 Gradle 构建配置文件

基于 Kotlin 来写 Gradle 构建脚本及插件的方式可能会是什么样的？它对团队（尤其是大型团队）帮助更大，可加快工作速度并编写结构更好、更易于维护的构建脚本。这些可能性非常诱人。

因为 Kotlin 是一种静态类型语言，在 IDEA 和 Eclipse 中都有深入的支持，所以可以从自动补全到重构，以及其间的一切都能为 Gradle 用户提供适当的 IDE 支持。而且由于 Kotlin 具有丰富的功能，如一等函数和扩展方法，因此它可以保留和改进 Gradle 构建脚本的最佳部分——包括简明的声明式语法以及轻松制作 DSL 的能力。

下面我们就简要介绍一下使用 Kotlin 脚本来编写 Gradle 的配置文件。

我们就以上一章中的 chapter11_kotlin_springboot 项目为例。

首先我们在根目录下新建一个 settings.gradle 配置文件：

```
rootProject.name = 'chapter11_kotlin_springboot'
rootProject.buildFileName = 'build.gradle.kts'
```

指定 gradle 构建文件名是‘build.gradle.kts’，然后，新建这个文件，完整的内容如下：

```
buildscript {
    val kotlinVersion = "1.1.3-2"
    val springBootVersion = "2.0.0.M2"
    extra["kotlinVersion"] = kotlinVersion

    repositories {
        mavenCentral()
        maven { setUrl("https://repo.spring.io/snapshot") }
        maven { setUrl("https://repo.spring.io/milestone") }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:
            $springBootVersion")
        classpath("org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlinVersion")
        classpath("org.jetbrains.kotlin:kotlin-allopen:$kotlinVersion")
    }
}

apply {
    plugin("kotlin")
    plugin("kotlin-spring")
    plugin("eclipse")
    plugin("org.springframework.boot")
    plugin("io.spring.dependency-management")
}

version = "0.0.1-SNAPSHOT"

configure<JavaPluginConvention> {
```

```
        setSourceCompatibility(1.8)
        setTargetCompatibility(1.8)
    }

    repositories {
        mavenCentral()
        maven { setUrl("https://repo.spring.io/snapshot") }
        maven { setUrl("https://repo.spring.io/milestone") }
    }

    val kotlinVersion = extra["kotlinVersion"] as String

    dependencies {
        compile("org.springframework.boot:spring-boot-starter-actuator")
        compile("org.springframework.boot:spring-boot-starter-data-jpa")
        compile("org.springframework.boot:spring-boot-starter-freemarker")
        compile("org.springframework.boot:spring-boot-starter-web")
        compile("org.jetbrains.kotlin:kotlin-stdlib-jre8:${kotlinVersion}")
        compile("org.jetbrains.kotlin:kotlin-reflect:${kotlinVersion}")
        runtime("org.springframework.boot:spring-boot-devtools")
        runtime("mysql:mysql-connector-java")
        testCompile("org.springframework.boot:spring-boot-starter-test")
    }
```



提示 按照上述步骤，新建完文件 build.gradle.kts 后，IDEA 可能识别不了这些 DSL 函数，这个时候我们重启一下 IDEA 即可（这是一个 bug，后面会修复）。

这里面与 Gradle DSL 相关的函数与类都在 Gradle 软件包的 lib 目录下：lib/gradle-script-kotlin-（版本号）.jar，参见表 12-1。

表 12-1 Gradle DSL 的相关函数与类

函数（类）	对应的 gradle-script-kotlin 代码
buildscript	`open fun buildscript(@Suppress("unused_parameter") block: ScriptHandlerScope.() -> Unit) = Unit`
repositories	`fun ScriptHandler.repositories(configuration: RepositoryHandler.() -> Unit) = repositories.configuration()`
buildscript.dependencies	`DependencyHandlerScope(scriptHandler.dependencies)`
configure	`inline fun <reified T: Any> Project.configure(noinline configuration: T.() -> Unit)`
Project.dependencies	`DependencyHandlerScope(dependencies).configuration()`

也就是说，其实这些配置函数背后都是由 Gradle 的 DSL 来实现的。

其实，这些配置语法看起与 Groovy 的很像。例如仓库配置：

Groovy：

```
repositories {
    mavenCentral()
    maven { url "https://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/milestone" }
}
```

Kotlin：

```
repositories {
    mavenCentral()
    maven { setUrl("https://repo.spring.io/snapshot") }
    maven { setUrl("https://repo.spring.io/milestone") }
}
```

编译 JDK 版本配置：

Groovy：

```
sourceCompatibility = 1.8
targetCompatibility = 1.8
```

Kotlin：

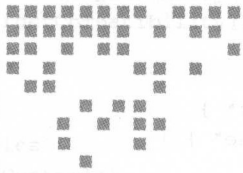
```
configure<JavaPluginConvention> {
    setSourceCompatibility(1.8)
    setTargetCompatibility(1.8)
}
```

本节示例代码：https://github.com/EasyKotlin/chapter12_gradle_kotlin_script

12.3　本章小结

本章简要介绍了使用 Kotlin 集成 Gradle 开发过程中的一些常用配置方法。Gradle 是一个非常好用的构建工具，当我们的 Kotlin 项目的配置文件也是 Kotlin 代码的时候，我们的工作又更加单纯了许多，只需要专注 Kotlin 即可。

在下一章中，我们将学习如何使用 Kotlin 和 Anko 进行 Android 开发。



Chapter 13 第 13 章

使用 Kotlin 和 Anko 的 Android 开发

Google 在今年（2017 年 5 月）的 IO 大会上宣布，将 Android 开发的官方语言更换为 Kotlin。通过前面内容的学习，我们可以体会到相对于 Java 来说，Kotlin 在编写代码时有如下优势：强大的扩展功能、代码简洁高效、函数式编程、空指针安全、支持 lambda 表达式、流式 API 等。

Kotlin 作为 Java 的改良，在 Android 开发中也有很多优势，本章通过一个具体的开发实例介绍使用 Kotlin 开发 Android 应用的具体方法。其中用到了 Anko、ButterKnife、Realm 等相关框架。

13.1 什么是 Anko

Anko（https://github.com/Kotlin/anko）是一个用 Kotlin 写的 Android DSL（Domain-Specific Language）。长久以来，Android 视图都是用 XML 来完成布局的。这些 XML 可重用性比较差。同时，在运行的时候，XML 要转换成 Java 表述，这在一定程度上占用了 CPU 和耗费了电量。

Anko 是一个 Kotlin 库，它使 Android 应用程序的开发变得更快、更容易，使代码干净、易于阅读，彻底摆脱繁杂的 Android XML 视图文件。

Anko 由几个部分组成，见表 13-1。

表 13-1 Anko 的模块及功能

模　块	功能说明
Anko Commons	使得对 intents、dialogs、logging 等操作更加简单的轻量级库
Anko Layouts	快速和类型安全的动态 Android 布局库

（续）

模　块	功能说明
Anko SQLite	用于 android sqlite 的查询 dsl 和分析库
Anko Coroutines	基于 kotlinx 的协程库

有了 Anko 我们就能直接用 Kotlin 在任何的 Activity、Fragment 或者 AnkoComponent 里来编写视图。

这里是一个转换成 Anko 的简单 XML 文件：

```
XML
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent">
    <EditText
        android:id="@+id/todo_title"
        android:layout_width="match_parent"
        android:layout_heigh="wrap_content"
        android:hint="@string/title_hint" />
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/add_todo" />
</LinearLayout>
```

用 Anko 描述同样的视图代码如下：

```
verticalLayout {
    var title = editText {
        id = R.id.todo_title
        hintResource = R.string.title_hint
    }
    button {
        textResource = R.string.add_todo
        onClick { view -> {
                //可以在这里添加一些处理逻辑
                title.text = "Foo"
            }
        }
    }
}
```

可以看到，在 button 布局中的 onClick 监听函数，因为我们使用 Kotlin 代码来设计视图，所以可以直接使用 title 变量（editText 视图对象）。

13.2　快速入门实例

下面通过一个“我的日程”待办事项应用，来详细介绍 Kotlin 混合 Java，使用 Anko 开发的 Android 应用的方法。移动端数据库引擎使用 Realm，视图绑定使用 Butter Knife。

13.2.1 功能介绍

这个应用程序基本功能是一个简单添加日程的界面、一个日程列表以及点击日程进行编辑保存的功能。它的主要界面原型设计如图 13-1 至图 13-3 所示。

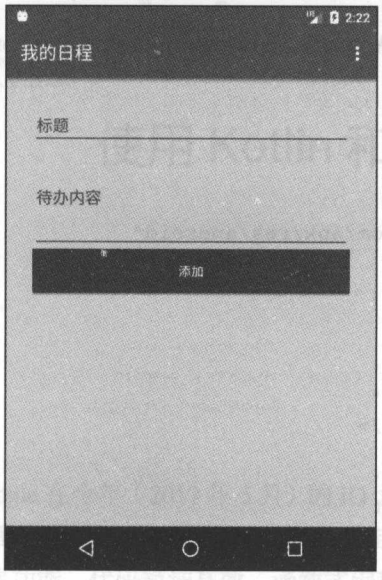


图 13-1 添加我的日程

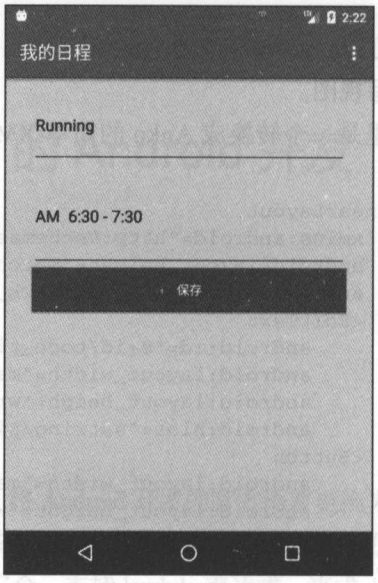


图 13-2 编辑日程

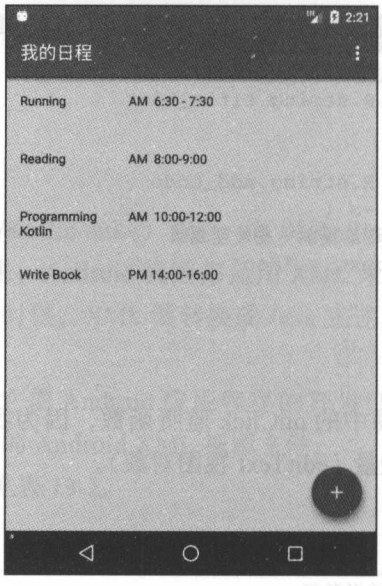


图 13-3 我的日程列表

下面我们就来详细介绍使用 Kotlin 以及相关的库来开发这个应用的全过程。

13.2.2　使用 Android Studio 新建项目

首先在 Android Studio 中新建项目，步骤如下：

第一步，新建项目

首先，打开菜单栏的 File → New Project，点击新建项目，如图 13-4 所示。

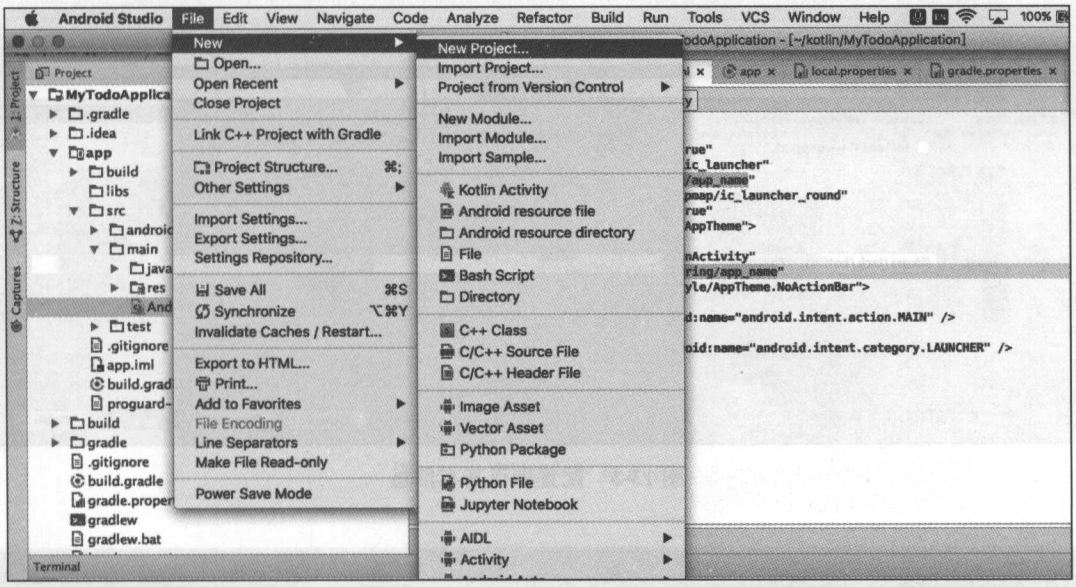


图 13-4　新建项目

第二步，配置项目基本信息

点击新建之后进入配置项目基本信息的界面，如图 13-5 所示。

输入相应的应用名称、组织域名、项目存放目录等信息。填完之后，点击 Next，进入 SDK 版本等配置界面。

第三步，设置支持设备以及 SDK 版本

在这个界面我们来配置 App 的运行环境以及 SDK 版本兼容性等选项。选择 Phone And Tablets，在 Minimum SDK 项目中选择“API 21：Android 5.0（Lollipop）”，如图 13-6 所示。

点击 Next，进入 Activity 选择界面。

第四步，选择 Basic Activity

直接选择 Basic Activity（基本 Activity），如图 13-7 所示。

点击 Next，进入 Activity 配置界面。

第五步，Activity 命名

直接用默认的 MainActivity 命名，如图 13-8 所示。

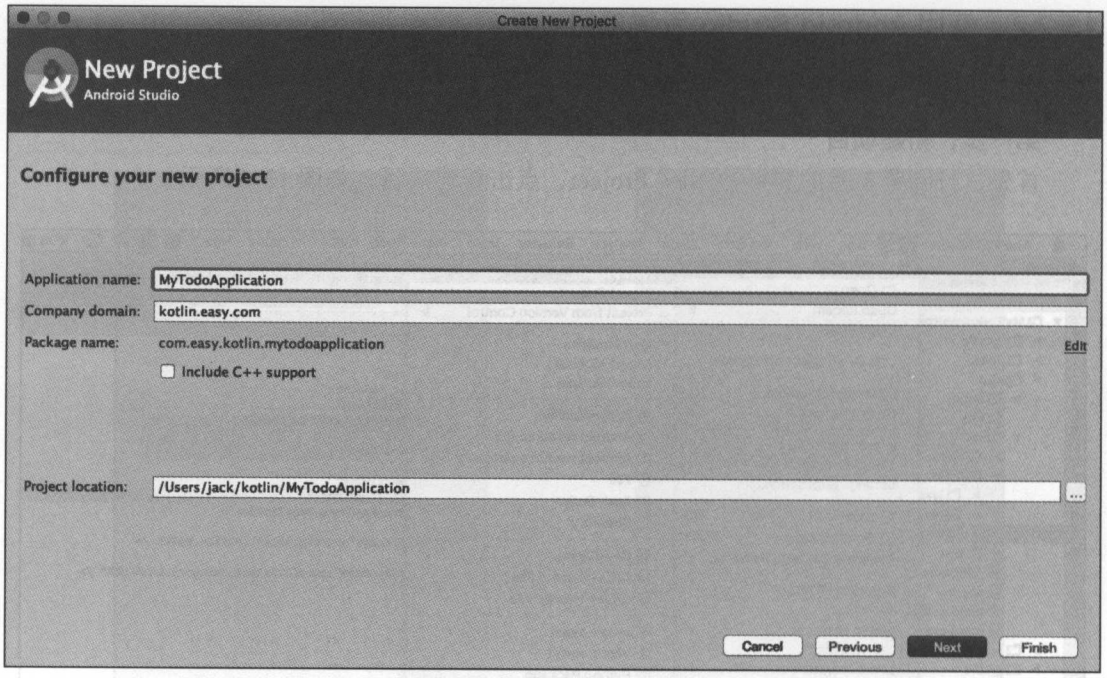


图 13-5 配置项目基本信息

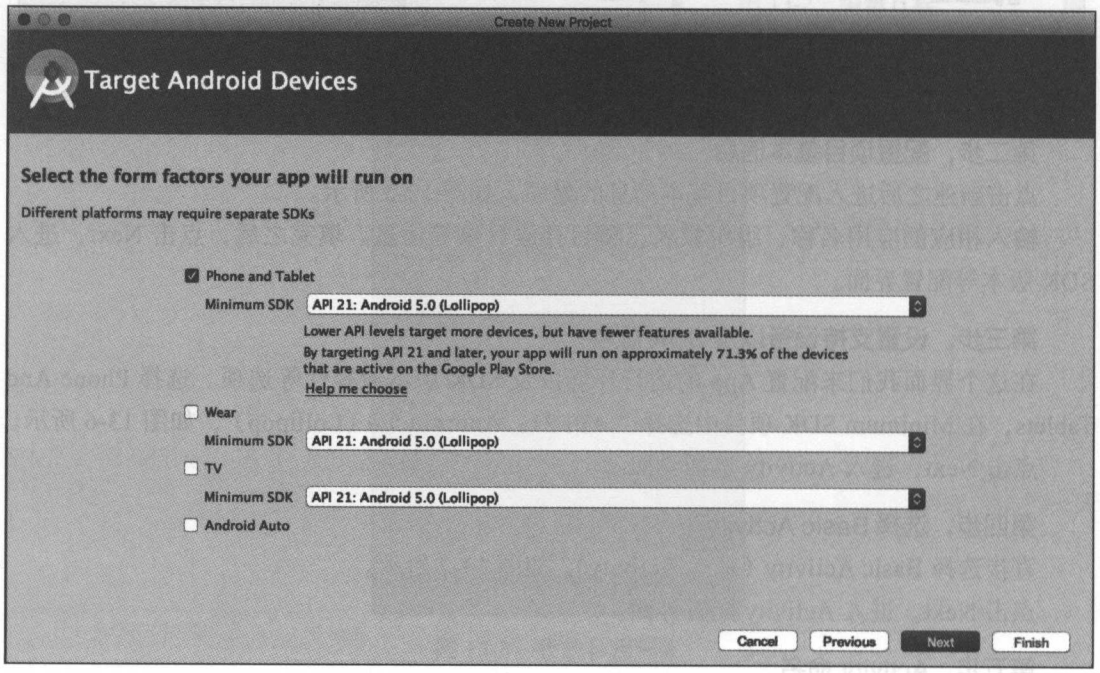


图 13-6 设置 SDK 版本等

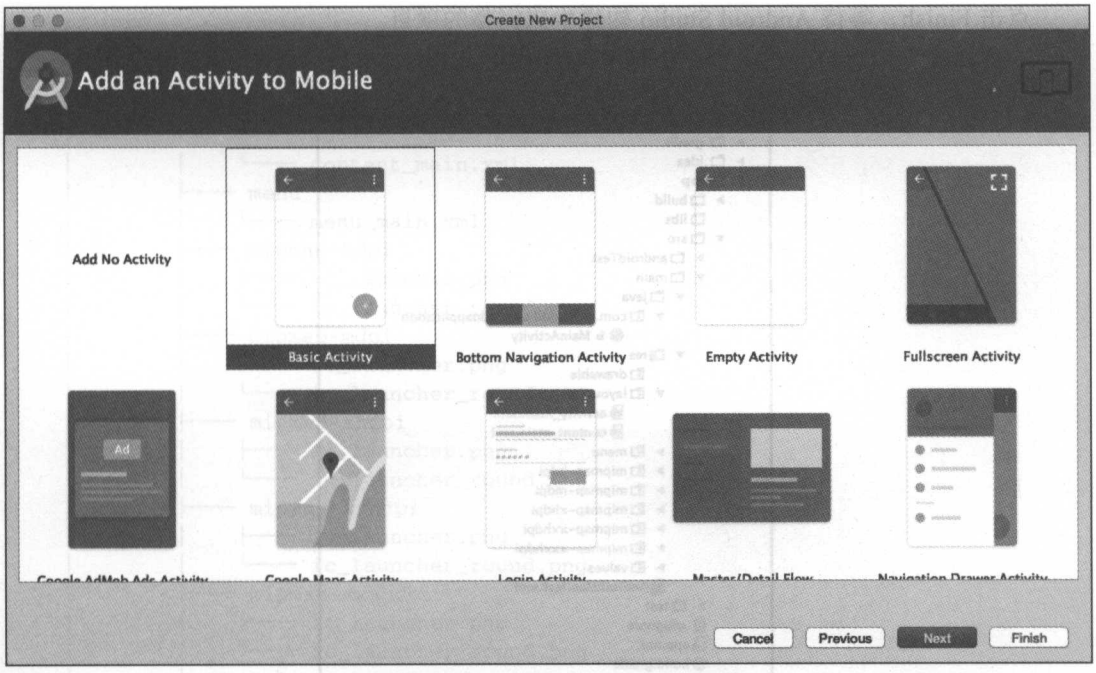


图 13-7　添加一个 Activity

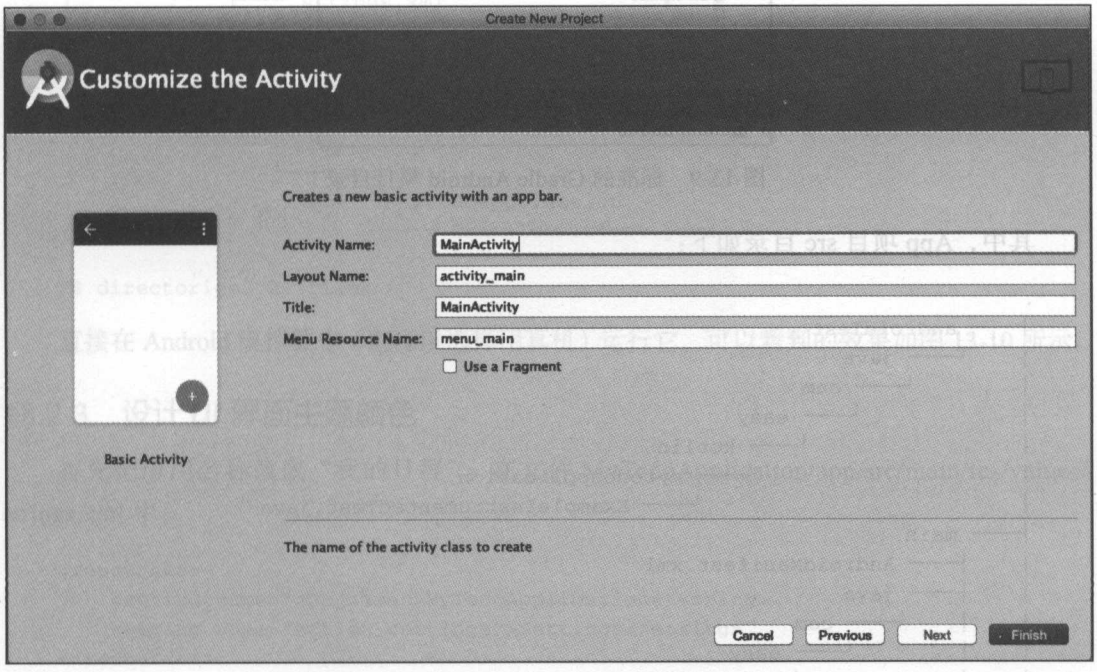


图 13-8　Activity 命名

点击 Finish，等待 Android Studio 构建生成完整的项目。

等待构建完毕之后，将得到一个标准的 Gradle Android 项目，如图 13-9 所示。

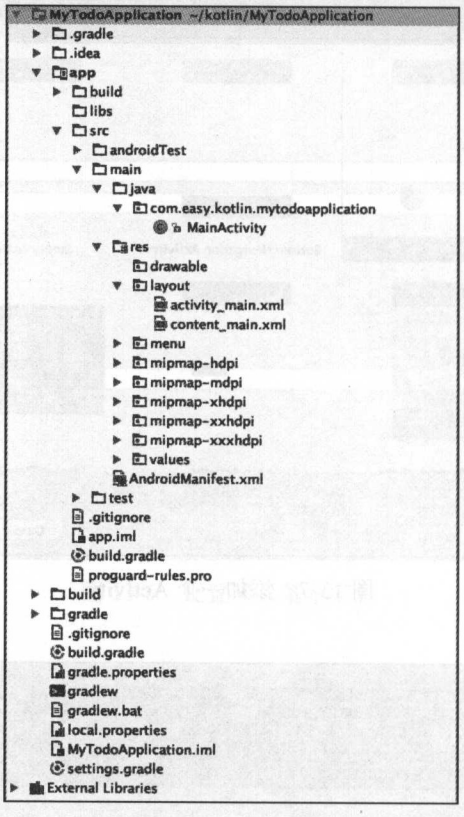


图 13-9 标准的 Gradle Android 项目目录

其中，App 项目 src 目录如下：

```
.
├── androidTest
│   └── java
│       └── com
│           └── easy
│               └── kotlin
│                   └── mytodoapplication
│                       └── ExampleInstrumentedTest.java
├── main
│   ├── AndroidManifest.xml
│   ├── java
│   │   └── com
│   │       └── easy
│   │           └── kotlin
│   │               └── mytodoapplication
```

```
│   │                   └── MainActivity.java
│   └── res
│       ├── drawable
│       ├── layout
│       │   ├── activity_main.xml
│       │   └── content_main.xml
│       ├── menu
│       │   └── menu_main.xml
│       ├── mipmap-hdpi
│       │   ├── ic_launcher.png
│       │   └── ic_launcher_round.png
│       ├── mipmap-mdpi
│       │   ├── ic_launcher.png
│       │   └── ic_launcher_round.png
│       ├── mipmap-xhdpi
│       │   ├── ic_launcher.png
│       │   └── ic_launcher_round.png
│       ├── mipmap-xxhdpi
│       │   ├── ic_launcher.png
│       │   └── ic_launcher_round.png
│       ├── mipmap-xxxhdpi
│       │   ├── ic_launcher.png
│       │   └── ic_launcher_round.png
│       └── values
│           ├── colors.xml
│           ├── dimens.xml
│           ├── strings.xml
│           └── styles.xml
└── test
    └── java
        └── com
            └── easy
                └── kotlin
                    └── mytodoapplication
                        └── ExampleUnitTest.java

28 directories, 21 files
```

直接在 Android 模拟器中（也可以选择用真机）运行它，可以看到的效果如图 13-10 所示。

13.2.3　设计 UI 界面主题颜色

首先把应用名称改成“我的日程”。在文件 MyTodoApplication/app/src/main/res/values/strings.xml 中：

```
<resources>
    <string name="app_name">MyTodoApplication</string>
    <string name="action_settings">Settings</string>
</resources>
```

改写成：

```
<resources>
    <string name="app_name">我的日程</string>
    <string name="action_settings">设置</string>
</resources>
```

再去 colors.xml 中设计主题颜色为：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#f2fced</color>
    <color name="colorPrimaryDark">#456a7c</color>
    <color name="colorAccent">#8fb3c4</color>
</resources>
```

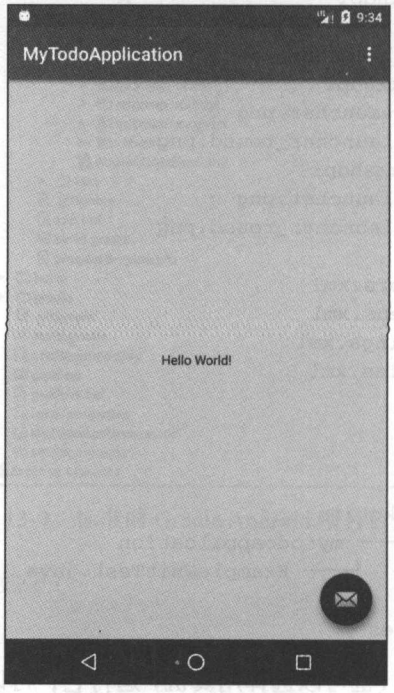


图 13-10 Hello, World

然后到文件 MyTodoApplication/app/src/main/res/layout/activity_main.xml 中，设置 android.support.v7.widget.Toolbar 的背景色为：

```
android:background="?attr/colorPrimaryDark"
```

配置 android.support.design.widget.FloatingActionButton 的图标为：

```
app:srcCompat="drawable/ic_content_add"
```

其中，ic_content_add.png 图片是我们添加按钮中间的加号 icon。

13.2.4　核心依赖配置

1. 配置 Kotlin 与 Anko 依赖

默认生成 App 项目的 Gradle 配置文件 build.gradle 如下：

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.3"
    defaultConfig {
        applicationId "com.easy.kotlin.mytodoapplication"
        minSdkVersion 21
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.3.1'
    compile 'com.android.support.constraint:constraint-layout:1.0.2'
    compile 'com.android.support:design:25.3.1'
    testCompile 'junit:junit:4.12'
}
```

下面我们在 App 项目的 build.gradle 里面加上 Kotlin、Anko、Realm、Butter Knife 等依赖。

2. 配置 Kotlin 依赖

首先，启用插件 kotlin-android：

```
apply plugin: 'kotlin-android'
```

然后，添加构建脚本：

```
buildscript {
}
```

使用 Kotlin 1.1.3 版本。在构建脚本中添加 kotlin-gradle-plugin 依赖，使用 Kotlin 对应的版本号：

```
buildscript {
    ext.kotlin_version = '1.1.3'
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

在项目依赖里添加 Kotlin 标准库：

```
// Kotlin
compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
```

3. 配置 Anko 依赖

在项目依赖里添加：

```
// Anko
compile 'org.jetbrains.anko:anko-sdk15:0.8.2' // sdk19, sdk21, sdk23 are also available
compile 'org.jetbrains.anko:anko-support-v4:0.8.2' // In case you need support-v4 bindings
compile 'org.jetbrains.anko:anko-appcompat-v7:0.8.2' // For appcompat-v7 bindings
```

4. 配置 Realm 依赖

在项目依赖里添加：

```
compile 'io.realm:realm-android:0.87.1'
compile 'com.github.thorbenprimke:realm-recyclerview:0.9.12' // 在 jitpack.io 上
```

其中，Realm 是一个轻量级的跨平台移动数据库引。Realm 简单易用，model 设计在代码中，更加易于维护，同时其性能也不错。在 Android 开发中，它可以替代 SQLite 和 ORM 框架。相比 SQLite，Realm 更快并且具有很多现代数据库的特性，比如支持 JSON、流式 API、数据变更通知、加密等。

RecyclerView 用于在有限的窗口展现大量的数据，相比 ListView、GridView，RecyclerView 标准化了 ViewHolder，而且更加灵活，可以轻松实现 ListView 实现不了的样式和功能。我们使用的 com.github.thorbenprimke:realm-recyclerview 依赖包在在 jitpack.io 上，所以还需要配置一下仓库地址：

```
repositories {
    mavenCentral()
    maven { url "https://jitpack.io" }
}
```



realm-recyclerview 的 Github 地址是 https://github.com/thorbenprimke/realm-recyclerview

另外，Kotlin 使用 Realm 还要加上注解处理的依赖库：

```
// kotlin 使用 realm 的注解处理依赖库
kapt "io.realm:realm-annotations:0.87.1"
kapt "io.realm:realm-annotations-processor:0.87.1"
```

5. 在 Kotlin 中使用 Realm

我们需要添加针对 Kotlin 的 realm 注解处理的库：

```
kapt "io.realm:realm-annotations:0.87.1"
kapt "io.realm:realm-annotations-processor:0.87.1"
```

6. 配置 Butter Knife 依赖

ButterKnife 是基于注解处理方式工作：通过对代码注解自动生成模板代码。我们添加其依赖如下：

```
// Butter Knife，专门为 Android View 设计的绑定注解，专业解决各种 findViewById
compile 'com.jakewharton:butterknife:8.7.0'
annotationProcessor 'com.jakewharton:butterknife-compiler:8.7.0'
```

Butter Knife 主要是用来做 Android 视图的成员变量和属性的数据绑定。在开发过程中，我们通常要写大量的 findViewById 和点击事件，像初始 view、设置 view 监听这样简单而重复的操作会显得比较繁琐。而有了 Butter Knife，就可以通过使用注解直接生成样板代码。例如，在 Java 中可以通过在字段上使用 @BindView 来替代 findViewById 的调用。上面的配置中，annotationProcessor 'com.jakewharton:butterknife-compiler:8.7.0' 就是来处理这些注解从而生成样板代码的：

```
@Bind(R.id.todo_item_todo_title)
public TextView todoTitle;

@Bind(R.id.todo_item_todo_content)
public TextView todoContent;
```

而在 Kotlin 中使用 Butter Knife 情况有些不同，需要作额外的配置。

如果在 Kotlin 中直接使用 Butter Knife 的注解方式，会出现空指针的异常，导致绑定失败。例如：

```
@Bind(R.id.todos_recycler_view)
var realmRecyclerView: RealmRecyclerView? = null
```

运行会报错：

```
Caused by: kotlin.KotlinNullPointerException
at com.easy.kotlin.mytodoapplication.TodoListFragment.onResume(TodoListFragment.
    kt:43)
```

一般情况下，我们使用 Kotlin 集成 Java 生态的一些框架的时候，像 Spring Boot、JPA、

Butter Knife、Realm 等，都需要一些额外的插件或者依赖来“填充缝隙”（例如：all-open，kotterknife、realm-annotations 等），所谓 Kotlin 与 Java 的无缝集成，很多时候并非 Java 中怎么用，Kotlin 就直接拿过来就怎么用，往往需要再添加一些插件或者额外的配置等。

那么要如何才能在 Kotlin 的环境中使用 ButterKnife 呢？

在早些时候，Butter Knife 的作者已经帮我们想好解决方案了，那就是 Kotter Knife，见名知意。Kotter Knife 的 GitHub 地址是：https://github.com/JakeWharton/kotterknife。这个插件是建立在 ButterKnife 7 的基础上的。

下面我们配置一下在 Kotlin 中使用 Butter Knife 的依赖库 Kotter Knife。

首先在 repositories 中添加 Kotter Knife 的仓库地址（Kotter Knife 不在 Maven Center 仓库中，而是在 oss.sonatype.org 仓库中。这么多仓库，要是哪天能统一就好了）：

```
repositories {
    ...
    maven { url 'https://oss.sonatype.org/content/repositories/snapshots/' }
}
```

然后在 dependencies 里面添加依赖：

```
dependencies {
...
compile 'com.jakewharton:butterknife:7.0.1'
compile 'com.jakewharton:kotterknife:0.1.0-SNAPSHOT'
}
```

采用这种方式的配置，我们的视图注入代码如下：

```
val todoTitle: TextView by bindView(R.id.todo_item_todo_title)
val todoContent: TextView by bindView(R.id.todo_item_todo_content)
```

这样的代码看起来不是那么的优雅，还没有在 Java 中直接使用注解来得简单好看。同时要注意的是，如果使用 kotterknife 0.1.0 + butterknife:7.0.1，同时使用 Java 跟 Kotlin 混合编程的场景中使用 Butter Knife，发现配了 Kotter Knife 之后的 Java 的注解式写法就失效了。也就是说，如果我们上面添加了 Kotter Knife 的依赖，那么 Java 代码中同时使用 Butter Knife 注解的地方会绑定失败。不过这个问题在后面的新版本中已经解决。例如在 butterknife 8.7.0 中，我们可以直接添加下面的依赖项：

```
compile 'com.jakewharton:butterknife:8.7.0'
annotationProcessor 'com.jakewharton:butterknife-compiler:8.7.0'
kapt 'com.jakewharton:butterknife-compiler:8.7.0'
```

其中，annotationProcessor 'com.jakewharton:butterknife-compiler:8.7.0' 是 Java 的 butterknife 注解处理器。kapt 'com.jakewharton:butterknife-compiler:8.7.0' 是 Kotlin 的 butterknife 注解处理器（Kotlin Annotation processing tool，kapt）。

这样我们的代码就继续优雅简洁下去了：

```
@BindView(R.id.todo_item_todo_title)
lateinit var todoTitle: TextView
@BindView(R.id.todo_item_todo_content)
lateinit var todoContent: TextView
```

其中，lateinit 修饰符允许声明非空类型，并在对象创建后（构造函数调用后）初始化。不使用 lateinit 则需要声明可空类型并且有额外的空安全检测操作。

当然，我们使用 Butter Knife 的同时，仍然可以使用原生的 findViewById：

```
class MainActivity : AppCompatActivity() {
    var fab: FloatingActionButton? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val toolbar = findViewById(R.id.toolbar) as Toolbar
        setSupportActionBar(toolbar)

        fab = findViewById(R.id.fab) as FloatingActionButton


        //添加日程事件
        fab?.setOnClickListener { _ ->
            ...
            hideFab()
        }
    }

    fun hideFab() {
        fab?.visibility = View.GONE
    }

    fun showFab() {
        fab?.visibility = View.VISIBLE
    }

}
```

13.2.5　代码编写

1. 添加 Kotlin 源代码目录

首先，在 src/main/ 下面新建一个 kotlin 目录，用于存放 Kotlin 源码。然后在 build.gradle 文件里的 android {} 配置里面添加 Java 的编译路径：

```
android {
    ...
    sourceSets {
```

```
        // += ，在 main 中创建 kotlin 文件夹，用于存放 kotlin 代码
        main.java.srcDirs += 'src/main/kotlin'
    }
}
```

刚添加完毕，src/main/kotlin 还没有变成源码目录的蓝色，这个时候点击图右上角的 Sync Now，见图 13-11。

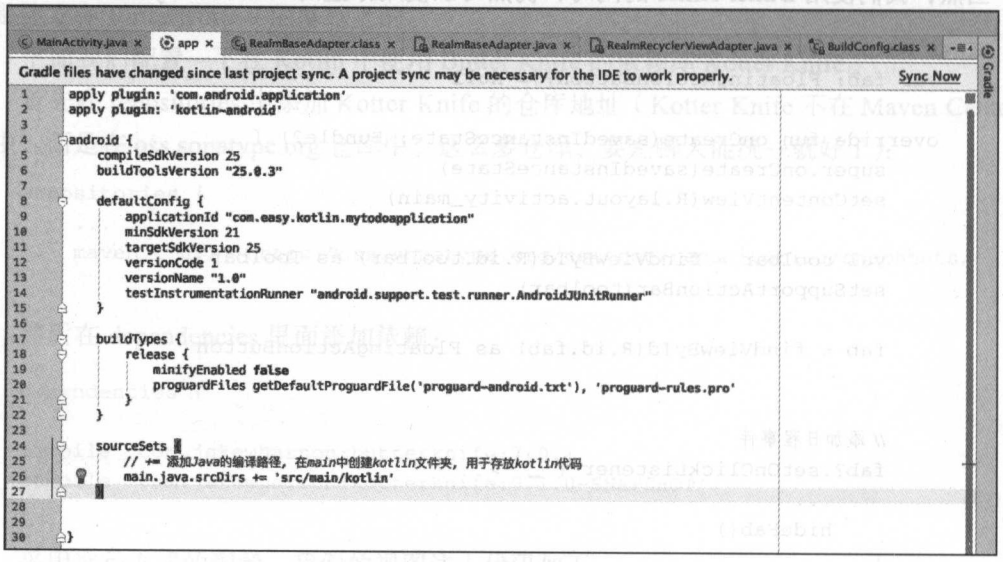


图 13-11 Gradle Sync

Gradle 同步完毕，即可看到 kotlin 目录已经变成蓝色的源码目录了，见图 13-12。

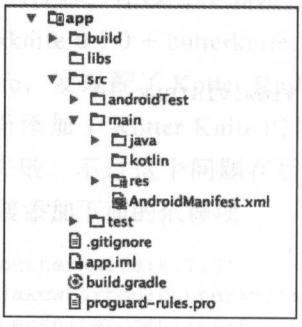


图 13-12 源码目录

2. 将 MainActivity. java 转成 Kotlin 代码

选中默认生成的 MainActivity.java，我们使用 IDEA 的 Code → Convert Java File to Kotlin File，如图 13-13 所示。



图 13-13　IDEA 的 Code 选项

点击转换，即可看到转换成 Kotlin 的代码：

```
package com.easy.kotlin.mytodoapplication

import android.os.Bundle
import android.support.design.widget.FloatingActionButton
import android.support.design.widget.Snackbar
import android.support.v7.app.AppCompatActivity
import android.support.v7.widget.Toolbar
import android.view.Menu
import android.view.MenuItem

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toolbar = findViewById(R.id.toolbar) as Toolbar
        setSupportActionBar(toolbar)

        val fab = findViewById(R.id.fab) as FloatingActionButton
        fab.setOnClickListener { view ->
            Snackbar.make(view, "Replace with your own action", Snackbar.
                LENGTH_LONG)
                    .setAction("Action", null).show()
        }
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
```

```
        // Inflate the menu; this adds items to the action bar if it is present.
        menuInflater.inflate(R.menu.menu_main, menu)
        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        val id = item.itemId


        if (id == R.id.action_settings) {
            return true
        }

        return super.onOptionsItemSelected(item)
    }
}
```

看，这就是 Android 开发者从 Java 无缝转到 Kotlin 的过程。

我们把这个 MainActivity.kt 放到对应的 src/main/kotlin 目录下。首先新建 package com.easy.kotlin.mytodoapplication，直接在 IDEA 中把这个 MainActivity.kt 拖到这个 package 下面即可。现在我们的项目目录如图 13-14 所示。

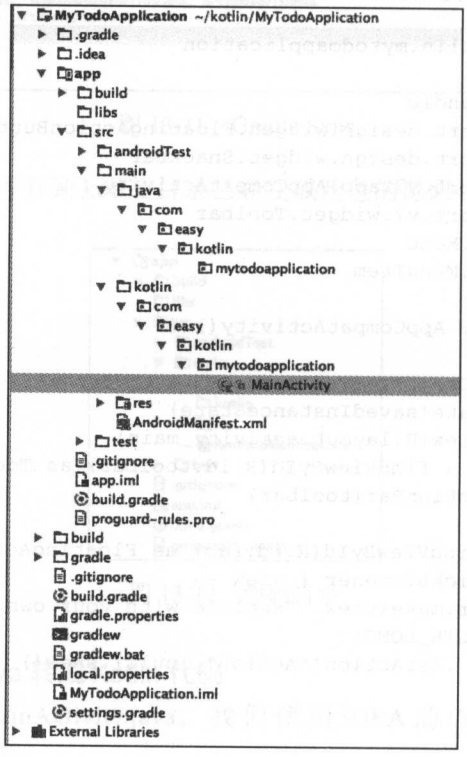


图 13-14 项目目录

3. 添加日程实体类

我们先从领域模型的建立开始。首先需要设计一个极简的待办事项的实体类 Todo，它有主键 id、标题、内容三个字段。

```
@RealmClass
open class Todo : RealmObject() {
    @PrimaryKey
    open var id: String = "-1"
    open var title: String = "日程"
    open var content: String = "事项"
}
```

然后，写一个应用程序入口类 MyTodoApplication 继承 android.app.Application，在 onCreate() 里面初始化 Realm 数据库的配置，代码如下：

```
class MyTodoApplication : Application() {
    override fun onCreate() {
        super.onCreate()

        val config = RealmConfiguration.Builder(this)
                .name("realm.my_todos")          //库文件名
                .encryptionKey(getKey())         //加密
                .schemaVersion(1)                //版本号
                .deleteRealmIfMigrationNeeded()
                .build()

        Realm.setDefaultConfiguration(config) //设置默认 RealmConfiguration

    }

    /**
     * 64 bits
     * @return
     */
    private fun getKey(): ByteArray {
        return byteArrayOf(0, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 4, 3, 2, 1, 0,
            1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 4, 3, 2,
            1, 0, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 4,
            3, 2, 1)
    }
}
```

RealmConfiguration.Builder 里面如果没有 deleteRealmIfMigrationNeeded() 的话，会如下报错误：

```
Caused by: io.realm.exceptions.RealmMigrationNeededException:
RealmMigration must be provided ...
at com.easy.kotlin.mytodoapplication.TodoListFragment.onActivityCreated
    (TodoListFragment.kt:36)
```



更多关于 Realm 数据库的相关内容可参考 https://realm.io/docs/

4. 添加日程事件

现在我们点击添加日程的浮层按钮中，添加切换到“日程添加编辑”TodoEditFragment 的逻辑：

```
//添加日程事件
fab?.setOnClickListener { _ ->
    //Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_
    //LONG).setAction("Action", null).show()
    val todoEditFragment = TodoEditFragment()
    getSupportFragmentManager()
            .beginTransaction()
            .replace(R.id.content_main, todoEditFragment, todoEditFragment.
                javaClass.getSimpleName())
            .addToBackStack(todoEditFragment.javaClass.getSimpleName())
            .commit()

    hideFab()
}
```

5. 添加日程界面

使用 Anko 的 Android 视图 DSL 来完成添加日程界面，这个界面设计如图 13-15 所示。

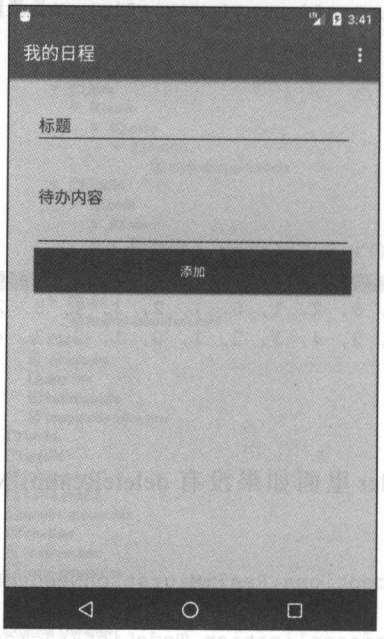


图 13-15 添加日程界面

下面我们就用 Anko DSL 来实现这个界面。

采用 Fragment 来实现。首先新建一个 TodoEditFragment 继承 Fragment()：

```
class TodoEditFragment : Fragment() {
    val realm: Realm = Realm.getDefaultInstance()
    var todo: Todo? = null

    companion object {
        val TODO_ID_KEY: String = "todo_id_key"

        fun newInstance(id: String): TodoEditFragment {
            var args: Bundle = Bundle()
            args.putString(TODO_ID_KEY, id)
            var todoEditFragment: TodoEditFragment = newInstance()
            todoEditFragment.arguments = args
            return todoEditFragment
        }

        fun newInstance(): TodoEditFragment {
            return TodoEditFragment()
        }
    }

    override fun onCreateView(inflater: LayoutInflater?, container: ViewGroup?,
        savedInstanceState: Bundle?): View? {
        return UI {
            // AnkoContext

            verticalLayout {
                padding = dip(30)
                var title = editText {
                    // editText 视图
                    id = R.id.todo_title
                    hintResource = R.string.title_hint
                }

                var content = editText {
                    id = R.id.todo_content
                    height = 400
                    hintResource = R.string.content_hint
                }
                button {
                    // button 视图
                    id = R.id.todo_add
                    textResource = R.string.add_todo
                    textColor = Color.WHITE
                    setBackgroundColor(Color.DKGRAY)
                    onClick { _ -> createTodoFrom(title, content) }
                }
            }
```

```
        }.view
    }

    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)

        if (arguments != null && arguments.containsKey(TODO_ID_KEY)) {

            val todoId = arguments.getString(TODO_ID_KEY)
            todo = realm.where(Todo::class.java).equalTo("id", todoId).findFirst()

            val todoTitle = find<EditText>(R.id.todo_title)
            todoTitle.setText(todo?.title)

            val todoContent = find<EditText>(R.id.todo_content)
            todoContent.setText(todo?.content)

            val add = find<Button>(R.id.todo_add)
            add.setText(R.string.save)
        }
    }

    override fun onDestroy() {
        super.onDestroy()
        realm.close()
    }


    /**
     *  新增待办事项，存入 Realm 数据库
     *
     *  @param title the title edit text.
     *  @param todoContent the content edit text.
     */
    private fun createTodoFrom(title: EditText, todoContent: EditText) {

        realm.beginTransaction()
        // Either update the edited object or create a new one.
        var t = todo ?: realm.createObject(Todo::class.java)
        t.id = todo?.id ?: UUID.randomUUID().toString()
        t.title = title.text.toString()
        t.content = todoContent.text.toString()

        realm.commitTransaction()
        activity.supportFragmentManager.popBackStack()
    }


}
```

其中，我们重点讲下 Anko 的 UI 布局部分的代码：

```
return UI {
    // AnkoContext

    verticalLayout {
        padding = dip(30)
        var title = editText {
            // editText 视图
            id = R.id.todo_title
            hintResource = R.string.title_hint
        }

        var content = editText {
            id = R.id.todo_content
            height = 400
            hintResource = R.string.content_hint
        }
        button {
            // button 视图
            id = R.id.todo_add
            textResource = R.string.add_todo
            textColor = Color.WHITE
            setBackgroundColor(Color.DKGRAY)
            onClick { _ -> createTodoFrom(title, content) }
        }
    }
}.view
```

使用 Kotlin 的代码 Anko DSL 创建一个垂直方向的线性布局（用代码写配置、布局要比 XML 灵活方便多了）。在 AnkoContext（UI 类）上调用 view 属性返回视图。其中，UI 函数如下所示：

```
fun Fragment.UI(init: AnkoContext<Fragment>.() -> Unit) = createAnkoContext
    (activity, init)
```

UI 函数是 Fragment 的扩展函数，它接收如下一个函数：

```
init: AnkoContext<Fragment>.() -> Unit
```

init 的入参是 AnkoContext 类型，而 verticalLayout 函数则是 ViewManager 的内联扩展函数：

```
inline fun ViewManager.verticalLayout(init: _LinearLayout.() -> Unit): LinearLayout {
    return ankoView(`$$Anko$Factories$CustomViews`.VERTICAL_LAYOUT_FACTORY, init)
}
```

从这些例子中我们可以看出，Kotlin 的函数扩展功能相当实用，尤其在 DSL 中用得非常广泛。

在 verticalLayout 代码段内部，创建了三个 Android 的控件：两个 editText 视图和一个 button 视图。这里视图的属性都在一行里面设置好了，如下所示：

```
padding = dip(30)
var title = editText {
    //editText 视图
    id = R.id.todo_title
    hintResource = R.string.title_hint
}

var content = editText {
    id = R.id.todo_content
    height = 400
    hintResource = R.string.content_hint
}
button {
    //button 视图
    id = R.id.todo_add
    textResource = R.string.add_todo
    textColor = Color.WHITE
    setBackgroundColor(Color.DKGRAY)
    onClick { _ -> createTodoFrom(title, content) }
}
```

这样的视图文件要比 XML 优雅多了。我们可以看到按钮 button{} 部分的代码，其中的 createTodoFrom 函数直接调用了外面的参数 title 和 content。这样的简单方便的调用方式在 XML 配置文件中是无法做到的。

这里的 ids 被设置为 R.id.<id_name>。这些 ids 需要手工在一个叫做 ids.xml 的文件里创建，这个文件放在 app/src/main/res/values/ids.xml。如果这个文件不存在就创建它。文件内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <item name="todo_title" type="id" />
    <item name="todo_content" type="id" />
    <item name="todo_add" type="id" />
</resources>
```

ids.xml 文件定义了所有能够被代码引用到的各种视图的 ids。

6. 数据持久化到 Realm 中

新增待办事项，存入 Realm 数据库，代码如下：

```
private fun createTodoFrom(title: EditText, todoContent: EditText) {

    realm.beginTransaction()
    //Either update the edited object or create a new one.
    var t = todo ?: realm.createObject(Todo::class.java)
```

```
        t.id = todo?.id ?: UUID.randomUUID().toString()
        t.title = title.text.toString()
        t.content = todoContent.text.toString()

        realm.commitTransaction()

        activity.supportFragmentManager.popBackStack()
    }
```

7. 用 RecyclerView 来展示待办事项

下面来实现“我的日程”列表界面，这个界面原型图设计如图 13-16 所示。

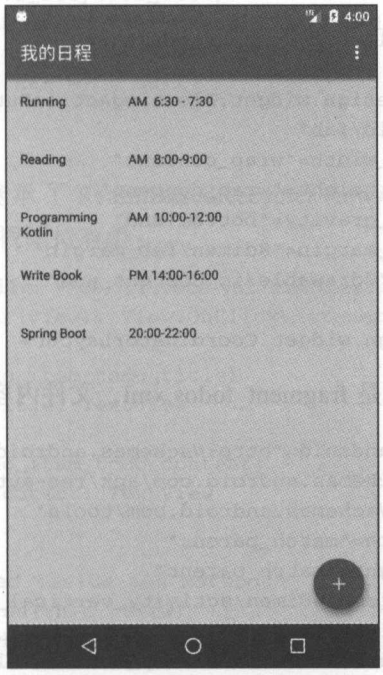


图 13-16　我的待办事项列表

首先，这是主页面，对应 activity_main.xml 视图，文件内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context="com.easy.kotlin.mytodoapplication.MainActivity">

    <android.support.design.widget.AppBarLayout
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimaryDark"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>

    <include layout="@layout/content_main" />

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        app:srcCompat="@drawable/ic_content_add" />

</android.support.design.widget.CoordinatorLayout>
```

我们的待办事项列表视图是 fragment_todos.xml，文件内容如下：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".TodosFragment"
    tools:showIn="@layout/activity_main">

    <co.moonmonkeylabs.realmrecyclerview.RealmRecyclerView
        android:id="@+id/todos_recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:rrvEmptyLayoutId="@layout/empty_view"
        app:rrvIsRefreshable="false"
        app:rrvLayoutType="LinearLayout" />

</RelativeLayout>
```

我们看一下 RealmRecyclerView 的配置，如表 13-2 所示。

表 13-2　RealmRecyclerView 的配置

配置项	功能说明
app:rrvEmptyLayoutId	当列表为空时的显示页面
app:rrvIsRefreshable	是否支持下拉刷新，通过 setOnRefreshListener 或 setRefreshing 来进行事件处理
app:rrvLayoutType	配置 LayoutManager，可选项是 LinearLayout、Grid、LinearLayoutWithHeaders 等

下面实现这个 TodosFragment。

首先新建 TodosFragment 类，继承如下面代码所示：

```
class TodosFragment : Fragment(), TodoAdapter.TodoItemClickListener {
    @BindView(R.id.todos_recycler_view)
    lateinit var realmRecyclerView: RealmRecyclerView

    private var realm: Realm? = null
    ...
}
```

其中，TodoAdapter 是继承了 RealmBasedRecyclerViewAdapter 的适配器类。我们在 TodoAdapter 里面定义了一个视图持有类：

```
inner class ViewHolder(view: View, private val clickListener: TodoItemClickListener?) :
        RealmViewHolder(view), View.OnClickListener {

    @BindView(R.id.todo_item_todo_title)
    lateinit var todoTitle: TextView

    @BindView(R.id.todo_item_todo_content)
    lateinit var todoContent: TextView

    init {
        // Bind annotated fields and methods
        ButterKnife.bind(this, view)
        view.setOnClickListener(this)
    }

    override fun onClick(v: View) {
        clickListener?.onClick(v, realmResults[adapterPosition])
    }
}
```

其中，@BindView 注解的功能是绑定 ID 为 todo_item_todo_title、todo_item_todo_content 的视图到变量 todoTitle、todoContent 上。

在 init 函数闭包中，使用 ButterKnife.bind 函数来绑定 view 变量到 ViewHolder 对象。

然后，定义一个待办事项监听器接口：

```
interface TodoItemClickListener {
    fun onClick(caller: View, todo: Todo)
}
```

在 TodosFragment 中实现这个方法如下：

```
override fun onClick(caller: View, todo: Todo) {
    (activity as MainActivity).hideFab()

    val todoEditFragment = TodoEditFragment.newInstance(todo.id)
    activity.supportFragmentManager
            .beginTransaction()
            .replace(R.id.content_main, todoEditFragment, todoEditFragment.
                javaClass.getSimpleName())
            .addToBackStack(todoEditFragment.javaClass.getSimpleName())
            .commit()
}
```

点击待办事项，把当前的 content_main 切换成编辑事项 EditFragment 的视图。

然后我们在 TodoAdapter 中重写 RealmBasedRecyclerViewAdapter 的 onCreateRealmViewHolder 和 onBindRealmViewHolder 方法：

```
override fun onCreateRealmViewHolder(viewGroup: ViewGroup, viewType: Int):
    ViewHolder {
    val v = inflater.inflate(R.layout.todo_item, viewGroup, false)
    return ViewHolder(v, clickListener)
}

override fun onBindRealmViewHolder(viewHolder: ViewHolder, position: Int) {
    val todo = realmResults[position]

    viewHolder.todoTitle.setText(todo.title)
    viewHolder.todoTitle.fontFeatureSettings = "font-size:12px"
    viewHolder.todoTitle.setTextColor(Color.argb(255, 69, 106, 124))

    viewHolder.todoContent.setText(todo.content)
}
```

添加、保存待办事项实现代码如下：

```
private fun createTodoFrom(title: EditText, todoContent: EditText) {
        realm.beginTransaction()
        // (1)
        var t = todo ?: realm.createObject(Todo::class.java)
        // (2)
        t.id = todo?.id ?: UUID.randomUUID().toString()
        t.title = title.text.toString()
        t.content = todoContent.text.toString()

        realm.commitTransaction()

        activity.supportFragmentManager.popBackStack()
    }
```

其中，代码注释部分简单说明如下：

- 当 todo 为空，通过 realm 的 createObject 函数新建一个 Todo 对象赋值给变量 t；否则把 todo 直接赋值给 t
- 当 todo 为空，通过 UUID 的随机算法新生成一个主键 id 赋值给 todo.id；否则直接把 todo 的 id 赋值给 t.id

在提交执行 realm 的事务之后，我们调用 popBackStack 函数返回到待办事项列表页。当回退到待办事项列表的时候，我们在 TodosFragment 中的 onResume() 函数中来实现数据的更新展示：

```
override fun onResume() {
    super.onResume()
    val todos = realm!!.where(Todo::class.java).findAll()
    Log.i(MY_TAG, "onResume: ${todos}")
    Log.i(MY_TAG, "onResume: realmRecyclerView = ${realmRecyclerView} ")
    val adapter = TodoAdapter(activity, todos, true, true, this)
    realmRecyclerView.setAdapter(adapter)
}
```

其中，val todos = realm!!.where(Todo::class.java).findAll() 是去 Realm 数据库中查询出所有 Todo 对应的实体记录。然后，通过适配器 val adapter = TodoAdapter(activity, todos, true, true, this) 把数据装配到 RecyclerView 中 realmRecyclerView.setAdapter(adapter)。

13.2.6　运行测试

先在 Android Studio 中打开模拟器，点击模拟器图标（图中工具栏中的最后一个），如图 13-17 所示。



图 13-17　我的待办事项列表

我们配置的设备信息如图 13-18 所示。点击执行按钮，启动该模拟器。

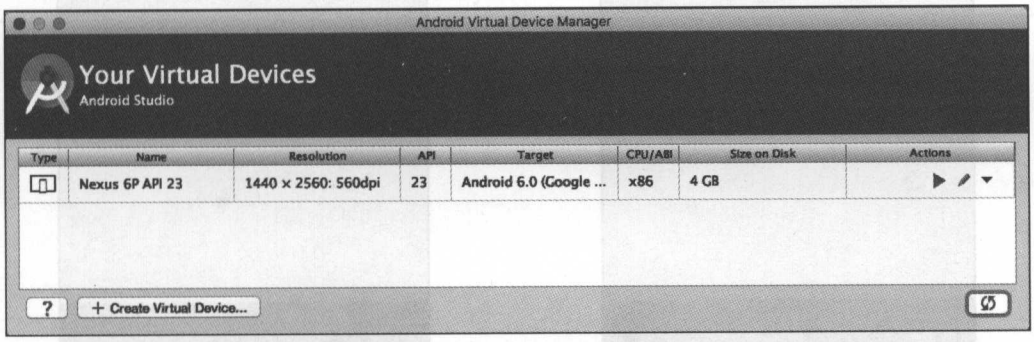


图 13-18　我的待办事项列表

模拟器成功启动后，界面如图 13-19 所示。

然后，在 Gradle 工具栏中点击 installDebug，如图 13-20 所示。

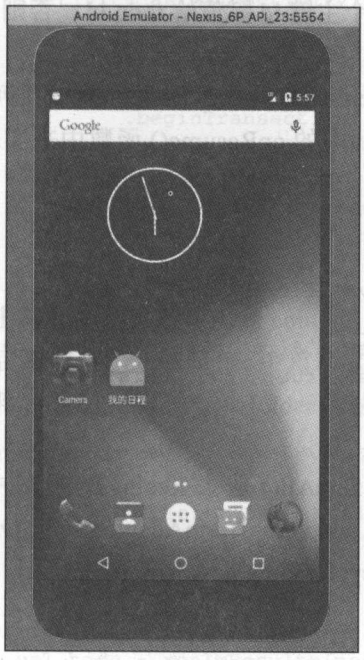


图 13-19 我的待办事项列表

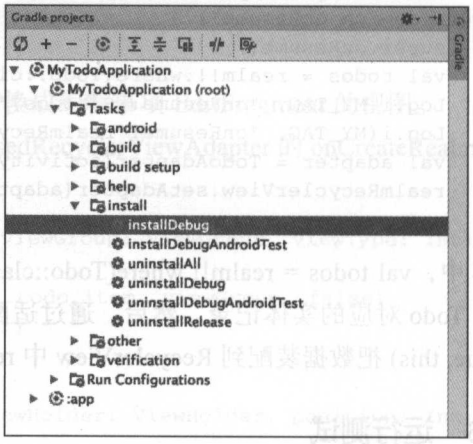


图 13-20 我的待办事项列表

直接编译安装应用，就可以看到如图 13-21 所示的界面了，现在可以在里面添加编辑我们的待办事项了，如图 13-22 和图 13-23 所示。

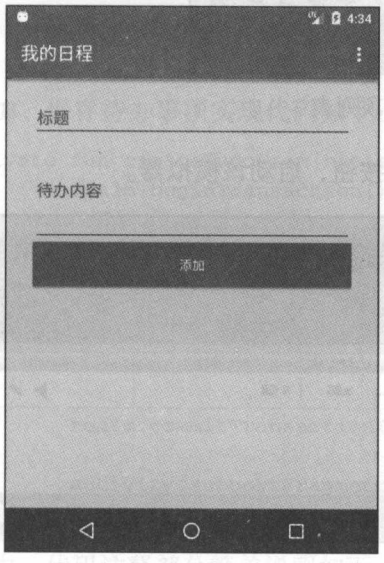


图 13-21 添加日程

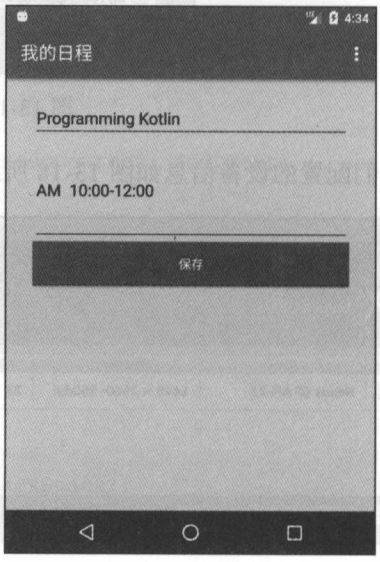


图 13-22 编辑日程

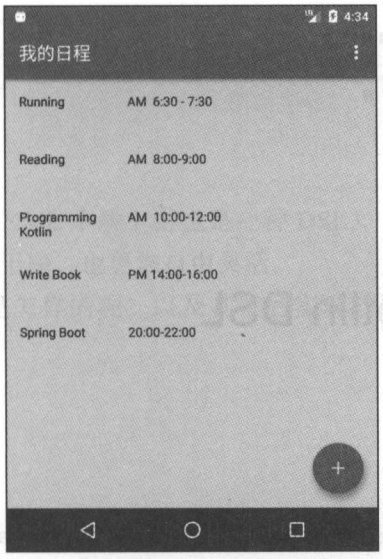


图 13-23　我的日程列表

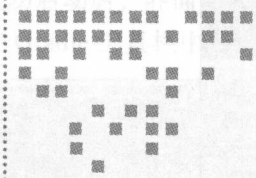
13.3　本章小结

Android 中经常出现的空引用、API 的冗余样板式代码等都是驱动我们转向 Kotlin 语言的动力。另外，Kotlin 的 Android 视图 DSL Anko 帮我们从繁杂的 XML 视图配置文件中解放出来。我们可以像在 Java 中一样方便的使用 Android 开发的流行的库诸如 Butter Knife、Realm、RecyclerView 等。当然，我们使用 Kotlin 集成这些库来进行 Andorid 开发，既能够直接使用我们之前的开发库，又能够从 Java 语言、Android API 的限制中解放出来，这不得不说是一件好事。

下一章我们介绍使用 Kotlin 创建 DSL。

本章示例代码：https://github.com/EasyKotlin/chapter13_kotlin_android

---



Chapter 14 第 14 章

使用 Kotlin DSL

前面的章节中已经介绍了 Kotlin DSL 的强大功能，例如 Gradle 的配置文件 build.gradle（Groovy），以及 Gradle Script Kotlin（Kotlin）、Anko（Kotlin）等，都是 DSL。我们可以看出，使用 DSL 的编程风格，可以让程序更加简单干净、直观简洁。当然，我们也可以创建自己的 DSL。本章就介绍 Kotlin 中 DSL 的相关内容。

我们在上一章中已经看到了在 Android 中使用下面这样的嵌套 DSL 风格的代码来替代 XML 式风格的视图文件：

```
UI {
    // AnkoContext

    verticalLayout {
        padding = dip(30)
        var title = editText {
            // editText 视图
            id = R.id.todo_title
            hintResource = R.string.title_hint
        }

        var content = editText {
            id = R.id.todo_content
            height = 400
            hintResource = R.string.content_hint
        }
        button {
            // button 视图
            id = R.id.todo_add
            textResource = R.string.add_todo
```

```
                textColor = Color.WHITE
                setBackgroundColor(Color.DKGRAY)
                onClick { _ -> createTodoFrom(title, content) }
            }
        }
    }
```

相比 XML 风格的 DSL（XML 本质上讲也是一种 DSL），明显使用原生的编程语言（例如 Kotlin）DSL 风格更加简单干净，也更加自由灵活。

Kotlin DSL 的编程风格是怎样的呢？以及其背后实现的原理是怎样的呢？下面就让我一起来探讨一下。

14.1　什么是 DSL

DSL（Domain-Specific Language，领域特定语言）指的是专注于特定问题领域的计算机语言（或称为领域专用语言）。不同于通用的计算机语言（GPL），领域特定语言只用在某些特定的领域。

“领域”可能特指某一产业，比如保险、教育、航空、医疗等，也可能是指一种方法或者技术，比如 JavaEE、.NET、数据库、服务、消息、架构以及领域驱动设计。开发 DSL 语言的目的在于，我们要以一种更优雅、更简洁的方式面对领域中的一些列挑战。之所以能够这样，是因为这个语言刚好够用于解决领域中唯一存在的一系列挑战，一点儿不多、也一点儿不少。

比如用来显示网页的 HTML 语言，更加典型的例子是 Gradle，它基于 Ant 和 Maven，使用基于 Groovy 的 DSL 来声明项目构建配置 build.gradle，而不是传统的 XML。

简单讲，DSL 就是对一个特定问题进行更高层次的抽象表达（领域特定），使其更加简单易懂（更加容易理解的语义以及清晰的语义模型）。在软件开发中，我们经常会在不同的层面上建立抽象，并处理它们。建立抽象最常见的方式是实现一个程序库或框架。操纵框架最常见的方式是通过命令 / 查询式 API 调用。从这种角度来看，DSL 就是这个程序库的前端，但它提供了一种不同于命令 / 查询式 API 风格的操作方式。在这样的上下文中，程序库成了 DSL 的“语义模型”。因此，DSL 经常伴随着程序库出现。事实上，对于构建良好的 DSL 而言，语义模型是一个不可或缺的附属物。

DSL 只是问题解决方案模型的外部封装，这个模型可能是一个 API 库，也可能是一个完整的框架等。DSL 提供了思考特定领域问题的模型语言，这使得我们可以更加简单高效地来解决问题。DSL 聚焦一个特定的领域，简单易懂，功能极简但完备。DSL 让我们理解和使用模型更加简易。

DSL 有内部 DSL 跟外部 DSL 之分，Gradle、Anko 等都是我们使用通用编程语言（Java 和 Kotlin）创建的内部 DSL。

1. 内部DSL

内部 DSL 是指与项目中使用的通用目的编程语言（Java、C# 或 Ruby）紧密相关的一类 DSL，它基于通用编程语言实现。例如，Rails 框架被称为基于 Ruby 的 DSL，用于管理 Ruby 开发的 Web 应用程序。Rails 之所以被称为 DSL，原因之一在于 Rails 应用了一些 Ruby 语言的特性，使得基于 Rails 编程看上去与基于通用目的的 Ruby 语言编程并不相同。

根据 Martin Fowler 和 Eric Evans 的观点，一个框架或者程序库的 API 是否满足内部 DSL 的关键特征之一就是它是否有流式接口。因为使用流式接口我们能够用短小表达式去复合组成（嵌套）一个原本很长的表达式，使它读起来更加流畅自然。例如通常我们会写下面这样的代码，这是一个关于客户订单的简单操作代码

```
private void makeNormal(Customer customer) {
    Order o1 = new Order();
    customer.addOrder(o1);
    OrderLine line1 = new OrderLine(6, Product.find("TAL"));
    o1.addLine(line1);
    OrderLine line2 = new OrderLine(5, Product.find("HPK"));
    o1.addLine(line2);
    OrderLine line3 = new OrderLine(3, Product.find("LGV"));
    o1.addLine(line3);
    line2.setSkippable(true);
    o1.setRush(true);
}
```

在流式风格的接口中，代码将是这样子：

```
private void makeFluent(Customer customer) {
    customer.newOrder()
            .with(6, "TAL")
            .with(5, "HPK").skippable()
            .with(3, "LGV")
            .priorityRush();
}
```

2. 外部DSL

外部 DSL 与通用编程语言（GPL）类似，但是外部 DSL 更加专注于特定领域。

创建外部 DSL 和创建一种通用的编程语言的过程是相似的，它可以是编译型或者解释型的。它具有形式化的文法，只允许使用良好定义的关键字和表达式类型。经过编译的 DSL 通常不会直接产生可执行的程序（但是它确实可以）。

大多数情况下，外部 DSL 可以转换为一种与核心应用程序的操作环境相兼容的资源，也可以转换为用于构建核心应用的通用目的编程语言。例如，Hibernate 中使用的对象 – 关系映射文件，就是由外部 DSL 转换为资源的实例。



关于 DSL 的详细介绍可以参考：《领域特定语言》(Martin Fowler) 这本书。

14.2　Kotlin 的 DSL 特性支持

许多现代语言为创建内部 DSL 提供了一些先进的方法，Kotlin 也不例外。

我用本章开头示例的 UI{...} 代码简单说明一下 Kotlin 中的 DSL 特性，见表 14-1。

表 14-1　举例说明 Kotlin 中的 DSL

函数名	函数签名	备注说明
UI	fun Fragment.UI(init: AnkoContext.() -> Unit):AnkoContext	android.support.v4.app.Fragment 的扩展函数；入参 init 是一个带接收者的函数字面值，我们直接传入的是一个 Lambda 表达式
verticalLayout	inline fun ViewManager.verticalLayout (init:_LinearLayout.() -> Unit): LinearLayout	android.view.ViewManager 的扩展函数

在 Kotlin 中创建 DSL，一般使用下面三个特性：

- 扩展函数、扩展属性。
- 带接收者的函数字面值（Lambda 表达式、高阶函数等）。
- invoke 操作符函数调用约定。

关于扩展函数和带接收者的函数字面值在前面的章节中已经讲过了，下面简单讲一下 Kotlin 中的 invoke 操作符函数。

在前面的集合类章节中，我们知道 Kotlin 中使用下标运算符 foo[x] 来等价调用 foo.get(x) 操作符函数。同样，关于 invoke 操作符函数调用有一个类似的约定。

我们知道，对一个函数 predicate: (T) -> Boolean 可以直接这样调用 predicate(element)，代码实例可以在 List 的扩展函数 filterTo 中看到：

```
public inline fun <T, C : MutableCollection<in T>> Iterable<T>.filterTo(destination:
    C, predicate: (T) -> Boolean): C {
    for (element in this) if (predicate(element)) destination.add(element)
    return destination
}
```

在 Kotlin 中，可以将 foo.invoke() 简写成 foo()，在 kotlin 中操作符是可以重载的，() 操作符对应的就是类的重载操作符函数 invoke，即此处的 predicate(element) 等价于 predicate.invoke(element) 。

上面是函数类型对象的 invoke 函数例子。实际上在 Kotlin 中，在类的对象实例上也可以像函数那样直接使用 () 操作符来调用这个类的一个 invoke 操作符函数。用代码示例来说明可能会更加简单直接。一个简单的示例代码如下：

```
>>> class Hello{
...     operator fun invoke(name:String){
...         println("Hello, $name")
...     }
... }
```

```
>>> val hello = Hello()
>>> hello("World")
Hello, World
>>> hello("Kotlin")
Hello, Kotlin
```

这段代码在 Hello 类中定义了一个操作符函数 invoke，然后声明了一个 Hello 类的实例对象 hello。接下来，神奇的事情发生了：

```
hello("World")
```

我们直接把这个实例对象 hello 当做函数一样来调用了，给它传入了参数“World”，在 REPL 中运行上面的代码，我们发现正确输出了：

```
>>> hello("World")
Hello, World
```

一般情况下，这个特性在程序代码中很少使用到，但是在 DSL 中将会非常有用。这个特性会使得我们的 DSL 代码更加简洁清晰。

14.3 使用 kotlinx.html DSL 写前端代码

为了加深对 Kotlin DSL 实用性上的理解，本节再介绍一个 Kotlin 中关于 HTML 的 DSL：kotlinx.html。

kotlinx.html 是可在 Web 应用程序中用于构建 HTML 的 DSL。它可以作为传统模板系统（例如 JSP、FreeMarker 等）的替代品。

kotlinx.html 分别提供了 kotlinx-html-jvm 和 kotlinx-html-js 库的 DSL，用于在 JVM 和浏览器（或其他 javascript 引擎）中直接使用 Kotlin 代码来构建 html，代替了原有的 HTML 前端代码。使用 kotlinx.html 这个 Kotlin DSL 来写传统静态 HTML 页面，就像在 Android 中使用 Anko 这个 Kotlin DSL 来写 XML 视图一样，这将是一场奇妙的编程体验。



更多关于 kotlinx.html 的相关内容可以参考它的 Github 地址：https://github.com/Kotlin/kotlinx.html

要使用 kotlinx.html 首先添加依赖，如下所示：

```
dependencies {
    def kotlinx_html_version = "0.6.3"
    compile "org.jetbrains.kotlinx:kotlinx-html-jvm:${kotlinx_html_version}"
    compile "org.jetbrains.kotlinx:kotlinx-html-js:${kotlinx_html_version}"
    ...
}
```

kotlinx.html 最新版本发布在 https://jcenter.bintray.com/ 仓库上，所以我们添加一下仓库

的配置：

```
repositories {
    maven { url 'https://jitpack.io' }
    mavenCentral()
    jcenter() //https://jcenter.bintray.com/ 仓库
    maven { url "https://repo.spring.io/snapshot" }
    maven { url "https://repo.spring.io/milestone" }
}
```

我们来写一个极简百度首页示例。这个页面的前端 HTML 代码如下：

```
<!DOCTYPE html>
<html lang=zh-CN>
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name=viewport content="width=device-width,initial-scale=1">
    <title>百度一下</title>
    <link href="https://cdn.bootcss.com/bootstrap/4.0.0-alpha.6/css/bootstrap.
        min.css" rel="stylesheet">
    <script src="https://cdn.bootcss.com/jquery/3.2.1/jquery.min.js"></script>
    <link href="dsl.css" rel="stylesheet">
    <script src="dsl.js"></script>
</head>
<body>
<div class="container">
    <div class="ipad center">
        ![](http://upload-images.jianshu.io/upload_images/1233356-49a0fecdc8bfa9cf.
            png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)
    </div>

    <form class="form">
        <input id="wd" class="form-control ipad">
        <button id="baiduBtn" type="submit" class="btn btn-primary form-control
            ipad">百度一下</button>
    </form>
</div>
</body>
</html>
```

其中，dsl.css 文件内容如下所示：

```
.ipad {
    margin: 10px
}

.center {
    text-align: center;
}
```

dsl.js 文件内容如下：

```
$(function () {
    $('#baiduBtn').on('click', function () {
        var wd = $('#wd').val()
        window.open("https://www.baidu.com/s?wd=" + wd)
    })
})
```

上面是通常使用的 HTML+JS+CSS 方式来写前端页面的方法，现在我们把 HTML 部分的代码用 Kotlin 的 DSL kotlinx.html 来重新实现一遍。

我们首先新建 Kotlin + Spring Boot 项目，然后直接来写 Kotlin 视图类 HelloDSLView，代码如下：

```
package com.easy.kotlin.chapter14_kotlin_dsl.view

import kotlinx.html.*
import kotlinx.html.stream.createHTML
import org.springframework.stereotype.Service

/**
 * Created by jack on 2017/7/22.
 */
@Service
class HelloDSLView {
    fun html(): String {
        return createHTML().html {
            head {
                meta {
                    charset = "utf-8"
                    httpEquiv = "X-UA-Compatible"
                    content = "IE=edge"
                }
                title("百度一下")
                link {
                    href = "https://cdn.bootcss.com/bootstrap/4.0.0-alpha.6/
                        css/bootstrap.min.css"
                    rel = "stylesheet"
                }
                script {
                    src = "https://cdn.bootcss.com/jquery/3.2.1/jquery.min.js"
                }
                link {
                    href = "dsl.css"
                    rel = "stylesheet"
                }
                script {
                    src = "dsl.js"
                }
            }

            body {
                div(classes = "container") {
```

```
                div(classes = "ipad center") {
                    img {
                        src = "https://www.baidu.com/img/bd_logo1.png"
                        width = "270"
                        height = "129"
                    }
                }

                form(classes = "form") {
                    input(InputType.text, classes = "form-control ipad") {
                        id = "wd"
                    }
                    button(classes = "btn btn-primary form-control ipad") {
                        id = "baiduBtn"
                        type = ButtonType.submit
                        text(" 百度一下 ")

                    }
                }

            }
        }
    }
}
```

相比之下，我们使用 DSL 的风格要比原生 HTML 要简洁优雅。关键是，我们的这个 HTML 是用 Kotlin 写的，这也就意味着，HTML 代码不再是简单的静态的前端代码了。我们完全可以直接使用后端的接口返回数据来给 HTML 元素赋值，我们也完全具备了（当然是完全超越了）诸如 JSP、Freemarker 这样的视图模板引擎的各种判断、循环等的语法功能，因为我们直接使用的是一门强大的编程语言 Kotlin 来写的 HTML 代码。

然后，我们就可以直接在控制器层的代码里直接调用 Kotlin 视图代码了：

```
@Controller
class HelloDSLController {
    @Autowired
    var helloDSLView: HelloDSLView? = null

    @GetMapping("hello")
    fun helloDSL(model: Model): ModelAndView {
        model.addAttribute("hello", helloDSLView?.html())
        return ModelAndView("hello")
    }
}
```

为了简单起见，我们借用一下 Freemarker 来做视图解析引擎，但是它只负责原封不动地来传输我们的 Kotlin 视图代码。hello.ftl 代码如下：

```
${hello}
```

我们的源码目录如下：

```
src
├── main
│   ├── java
│   ├── kotlin
│   │   └── com
│   │       └── easy
│   │           └── kotlin
│   │               └── chapter14_kotlin_dsl
│   │                   ├── Chapter14KotlinDslApplication.kt
│   │                   ├── controller
│   │                   │   └── HelloDSLController.kt
│   │                   └── view
│   │                       └── HelloDSLView.kt
│   └── resources
│       ├── application.properties
│       ├── banner.txt
│       ├── static
│       │   ├── dsl.css
│       │   ├── dsl.js
│       │   └── hello.html
│       └── templates
│           └── hello.ftl
└── test
    ├── java
    ├── kotlin
    │   └── com
    │       └── easy
    │           └── kotlin
    │               └── chapter14_kotlin_dsl
    │                   └── Chapter14KotlinDslApplicationTests.kt
    └── resources
```

然后，启动运行 SpringBoot 应用，浏览器访问 http://127.0.0.1:8888/hello，我们可以看到输出界面，如图 14-1 所示。



图 14-1 使用 Kotlin DSL 绘制的极简百度首页

我们的这个极简百度首页没有写一行 HTML 代码，全部都是使用 Kotlin DSL 完成的。这就是 DSL 的精妙之处。后面可以尝试使用 kotlinx.html 来写 Kotlin 语言的前端代码。

在做 Web 开发的时候，我们通常是使用 HTML + 模板引擎（Velocity、JSP、Freemarker 等）来集成前后端的代码，这让我们有时候感到很尴尬，要学习模板引擎的语法，还得应对前端 HTML 代码中凌乱的模板引擎标签、变量等片段代码。

使用 Kotlin DSL 来写 HTML 代码的情况将完全不一样了，我们将重拾前后端集成编码的乐趣（不再是模板引擎套前端 HTML，各种奇怪的 #、<#>、${} 模板语言标签），我们直接把更加优雅简单的 DSL 风格的 HTML 代码搬到了后端，同时 HTML 中的元素将直接跟后端的数据无缝交互，而完成这些的只是 Kotlin（当然，相应领域的 DSL 基本语义模型还是要学习一下）。

本节示例代码：https://github.com/EasyKotlin/chapter14_kotlin_dsl

14.4　实现一个极简的 HTTP DSL

我们现在已经基本知道 Kotlin 中 DSL 的样子了。但是这些 DSL 都是怎样实现的呢？本节就通过实现一个极简的 HTTP DSL 来学习创建 DSL 背后的基本原理。

在这里我们对 OkHttp 做一下简单的封装，实现一个类似 jQuery 中的 Ajax 的 HTTP 请求的 DSL。

OkHttp 是一个成熟且强大的网络库，在 Android 源码中已经使用 OkHttp 替代原先的 HttpURLConnection。很多著名的框架例如 Picasso、Retrofit 也使用 OkHttp 作为底层框架。


提示　更多关于 OkHttp 的使用可参考：http://square.github.io/okhttp/

---

1. 创建 Kotlin Gradle 项目

我们首先使用 IDEA 创建 Kotlin Gradle 项目，如图 14-2 所示。

然后，在 build.gradle 里面配置依赖：

```
compile 'com.github.ReactiveX:RxKotlin:2.1.0'
compile group: 'com.squareup.okhttp3', name: 'okhttp', version: '3.8.1'
compile group: 'com.alibaba', name: 'fastjson', version: '1.2.35'
```

其中，RxKotlin 是 ReactiveX 框架对 Kotlin 语言的支持库。这里主要用 RxKotlin 来进行请求回调的异步处理。

我们使用的是‘com.github.ReactiveX:RxKotlin:2.1.0’，这个库是在 https://jitpack.io 上，所以我们在 repositories 配置里添加 jitpack 仓库：

```
repositories {
    maven { url 'https://jitpack.io' }
    ...
}
```

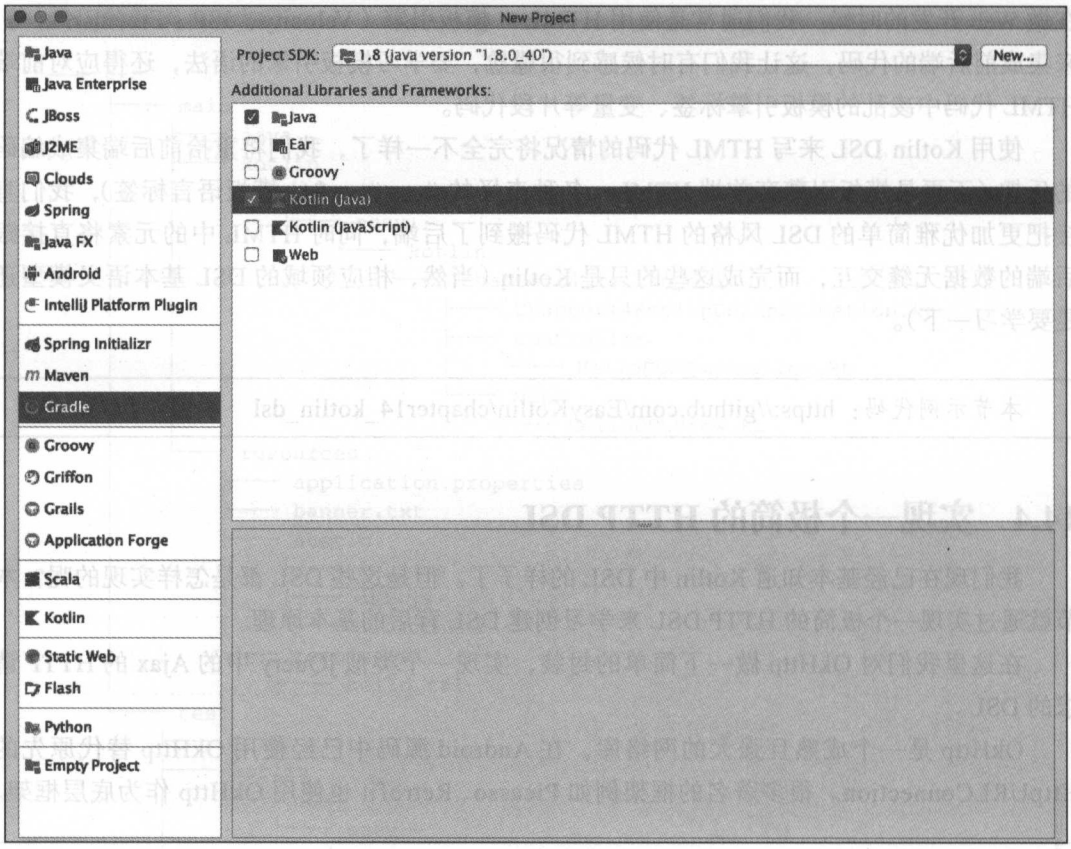


图 14-2 创建 Kotlin Gradle 项目

2. RxKotlin

ReactiveX 是 Reactive Extensions 的缩写，一般简写为 Rx，最初是 LINQ 的一个扩展，由微软的架构师 Erik Meijer 领导的团队开发，在 2012 年 11 月开源。

Rx 扩展了观察者模式用于支持数据和事件序列。Rx 是一个编程模型，目标是提供一致的编程接口，帮助开发者更方便地处理异步 I/O（非阻塞）数据流。

Rx 库支持 .NET、JavaScript 和 C++。Rx 近几年越来越流行，现在已经支持几乎全部的流行编程语言了，所支持的语言列表如表 14-2 所示。

表 14-2 Rx 支持的语言

Rx 支持的编程语言	项目主页
Java	RxJava：https://github.com/ReactiveX/RxJava
JavaScript	RxJS：https://github.com/ReactiveX/rxjs
C#	Rx.NET：https://github.com/Reactive-Extensions/Rx.NET
C#(Unity)	UniRx：https://github.com/neuecc/UniRx
Scala	RxScala：https://github.com/ReactiveX/RxScala
Clojure	RxClojure：https://github.com/ReactiveX/RxClojure

（续）

Rx 支持的编程语言	项目主页
C++	RxCpp：https://github.com/Reactive-Extensions/RxCpp
Lua	RxLua：https://github.com/bjornbytes/RxLua
Ruby	Rx.rb：https://github.com/Reactive-Extensions/Rx.rb
Python:	RxPY：https://github.com/ReactiveX/RxPY
Go	RxGo：https://github.com/ReactiveX/RxGo
Groovy	RxGroovy：https://github.com/ReactiveX/RxGroovy
JRuby	RxJRuby：https://github.com/ReactiveX/RxJRuby
Kotlin	RxKotlin：https://github.com/ReactiveX/RxKotlin
Swift	RxSwift：https://github.com/kzaher/RxSwift
PHP	RxPHP：https://github.com/ReactiveX/RxPHP
Elixir	reaxive：https://github.com/alfert/reaxive
Dart	RxDart：https://github.com/ReactiveX/rxdart

Rx 的大部分语言库由 ReactiveX 这个组织负责维护。Rx 比较流行的库有 RxJava/RxJS/Rx.NET 等，当然未来 RxKotlin 也必将更加流行。


提示　Rx 的社区网站是：http://reactivex.io/。Github 地址：https://github.com/ReactiveX/

3. HTTP 请求对象封装类

首先我们设计 HTTP 请求对象封装类如下：

```
class HttpRequestWrapper {

    var url: String? = null

    var method: String? = null

    var body: RequestBody? = null

    var timeout: Long = 10

    internal var success: (String) -> Unit = {}
    internal var fail: (Throwable) -> Unit = {}

    fun success(onSuccess: (String) -> Unit) {
        success = onSuccess
    }

    fun error(onError: (Throwable) -> Unit) {
        fail = onError
    }
}
```

HttpRequestWrapper 的成员变量和函数说明见表 14-3。

表 14-3 HttpRequestWrapper 的成员变量和函数

成 员	说 明
url	请求 url
method	请求方法，例如 Get、Post 等，不区分大小写
body	请求头，为了简单起见我们直接使用 OkHttp 的 RequestBody 类型
timeout	超时时间 ms，设置默认值是 10s
success	请求成功的函数变量
fail	请求失败的函数变量
fun success(onSuccess: (String) -> Unit)	请求成功回调函数
fun error(onError: (Throwable) -> Unit)	请求失败回调函数

4. HTTP 执行引擎

我们直接调用 OkHttp 的 HTTP 请求 API：

```
private fun call(wrap: HttpRequestWrapper): Response {

    var req: Request? = null
    when (wrap.method?.toLowerCase()) {
        "get" -> req = Request.Builder().url(wrap.url).build()
        "post" -> req = Request.Builder().url(wrap.url).post(wrap.body).build()
        "put" -> req = Request.Builder().url(wrap.url).put(wrap.body).build()
        "delete" -> req = Request.Builder().url(wrap.url).delete(wrap.body).build()
    }

    val http = OkHttpClient.Builder().connectTimeout(wrap.timeout, TimeUnit.
        MILLISECONDS).build()
    val resp = http.newCall(req).execute()
    return resp
}
```

它返回请求的响应对象 Response。

我们在 OkHttpClient.Builder().connectTimeout(wrap.timeout, TimeUnit.MILLISECONDS).build() 中设置超时时间的单位是 TimeUnit.MILLISECONDS。

我们通过 wrap.method?.toLowerCase() 处理请求方法的大小写的兼容。

5. 使用 RxKotlin 完成请求响应的异步处理

我们首先新建一个数据发射源：一个可观察对象（Observable），作为发射数据用：

```
val sender = Observable.create<Response>({
    e ->
    e.onNext(call(wrap))
})
```

其中，e 的类型是 io.reactivex.Emitter（发射器），它的接口定义是：

```
public interface Emitter<T> {
    void onNext(@NonNull T value);
    void onError(@NonNull Throwable error);
    void onComplete();
}
```

其方法功能简单说明如下：

方　法	功　能
onNext	发射一个正常值数据（value）
onError	发射一个 Throwable 异常
onComplete	发射一个完成的信号

这里，我们通过调用 onNext 方法，把 OkHttp 请求之后的响应对象 Response 作为正常值发射出去。

然后我们再创建一个数据接收源：一个观察者（Observer）：

```
val receiver: Observer<Response> = object : Observer<Response> {
    override fun onNext(resp: Response) {
        wrap.success(resp.body()!!.string())
    }

    override fun onError(e: Throwable) {
        wrap.fail(e)
    }

    override fun onSubscribe(d: Disposable) {
    }

    override fun onComplete() {
    }

}
```

receiver 的 onNext 函数接收 sender 发射过来的数据 Response，然后我们在函数体内，调用这个响应对象，给 wrap.success 回调函数进行相关的赋值操作。同样，onError 函数中也执行相应的赋值操作。

最后，通过 subscribe 订阅函数来绑定 sender 与 receiver 的关联：

```
sender.subscribe(receiver)
```

作为接收数据的 receiver（也就是观察者（Observer）），对发送数据的 sender（也就是可被观察对象（Observable））所发射的数据或数据序列作出响应。

这种模式可以极大地简化并发操作，因为它创建了一个处于待命状态的观察者，在未来某个时刻响应 sender 的通知，而不需要阻塞等待 sender 发射数据。这个很像协程中的通道编程模型。

6. DSL 主函数 ajax

我们的 ajax DSL 主函数设计如下：

```
fun ajax(init: HttpRequestWrapper.() -> Unit) {
    val wrap = HttpRequestWrapper()
    wrap.init()
    doCall(wrap)
}
```

其中，参数 init: HttpRequestWrapper.() -> Unit 是一个带接收者的函数字面量，它的类型是 init = Function1<com.kotlin.easy.HttpRequestWrapper, kotlin.Unit>。HttpRequestWrapper 是扩展函数 init() 的接收者，点号 . 是扩展函数修饰符。

我们在函数体内直接调用了这个函数字面量 wrap.init()。这样的写法可能比较难以理解，这个函数字面量 init 的调用实际上是 init.invoke(wrap)，就是把传入 ajax 的函数参数直接传递给 wrap。为了更简单的理解这个 init 函数的工作原理，我们通过把上面的 ajax 函数的代码反编译成对应的 Java 代码如下：

```
public static final void ajax(@NotNull Function1 init) {
    Intrinsics.checkParameterIsNotNull(init, "init");
    HttpRequestWrapper wrap = new HttpRequestWrapper();
    init.invoke(wrap);
    doCall(wrap);
}
```

也就是说，ajax 函数的一个更容易理解的写法是：

```
fun ajax(init: HttpRequestWrapper.() -> Unit) {
    val wrap = HttpRequestWrapper()
    init.invoke(wrap)
    doCall(wrap)
}
```

我们在实际应用的时候，可以直接把 init 写成 Lambda 表达式的形式，因为接收者类型 HttpRequestWrapper 可以从上下文推断出来。

我们这样调用 ajax 函数：

```
ajax {
    url = testUrl
    method = "get"
    success {
        string ->
        println(string)
        Assert.assertTrue(string.contains("百度一下"))
    }
    error {
        e ->
        println(e.message)
    }
}
```

下面是几个测试代码示例：

```
package com.kotlin.easy

import com.alibaba.fastjson.JSONObject
import okhttp3.MediaType
import okhttp3.RequestBody
import org.junit.Assert
import org.junit.Test
```

```
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

/**
 * Created by jack on 2017/7/23.
 */

@RunWith(JUnit4::class)
class KAjaxTest {

    @Test fun testHttpOnSuccess() {
        val testUrl = "https://www.baidu.com"
        ajax {
            url = testUrl
            method = "get"
            success {
                string ->
                println(string)
                Assert.assertTrue(string.contains("百度一下"))
            }
            error {
                e ->
                println(e.message)
            }
        }
    }

    @Test fun testHttpOnError() {
        val testUrl = "https://www2.baidu.com"

        ajax {
            url = testUrl
            method = "get"
            success {
                string ->
                println(string)
            }
            error {
                e ->
                println(e.message)
                Assert.assertTrue("connect timed out" == e.message)
            }
        }
    }

    @Test fun testHttpPost() {
        var json = JSONObject()
        json.put("name", "Kotlin DSL Http")
        json.put("owner", "Kotlin")
```

```
        val postBody = RequestBody.create(MediaType.parse("application/json;
            charset=utf-8"), json.toString())
        ajax {
            url = "saveArticle"
            method = "post"
            body = postBody
            success {
                string ->
                println(string)
            }
            error {
                e ->
                println(e.message)
            }
        }
    }


    @Test fun testLambda() {
        val testUrl = "https://www.baidu.com"

        val init: HttpRequestWrapper.() -> Unit = {
            this.url = testUrl
            this.method = "get"
            this.success {
                string ->
                println(string)
                Assert.assertTrue(string.contains("百度一下"))
            }
            this.error {
                e ->
                println(e.message)
            }
        }
        ajax(init)
    }
```

到这里，我们已经完成了一个极简的 Kotlin Ajax DSL。

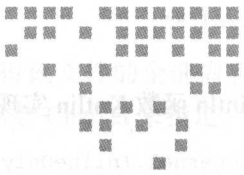
本节示例代码：https://github.com/EasyKotlin/chatper14_kotlin_dsl_http

---

14.5 本章小结

相比于 Java，Kotlin 对函数式编程的支持更加友好。Kotlin 的扩展函数和高阶函数（Lambda 表达式）为定义 Kotlin DSL 提供了核心的特性支持。

使用 DSL 的代码风格，可以让我们的程序更加直观易懂、简洁优雅。如果使用 Kotlin 来开发项目的话，我们完全可以去尝试一下。



第 15 章 Chapter 15

Kotlin 文件 IO 操作与多线程

我们在使用 Groovy 的文件 IO 操作时，感觉非常便利。同样，Kotlin 也有好用的文件 IO 操作 API。另外，Kotlin 对 Java 的正则表达式功能做了一些实用的扩展，Kotlin 中的多线程对 Java 的多线程 API 做了一些封装，因为 Java 已经有了很多的基础 API，Kotlin 并没有必要再去重复实现，而是在 Java 的基础上进行了实用的功能扩展。

本章就来介绍 Kotlin 文件 IO 操作、正则表达式以及多线程相关的内容。

15.1 Kotlin IO 基本操作

Kotlin 的 IO 操作都在 kotlin.io 包下。Kotlin 的原则就是 Java 已经有的、好用的就直接使用，没有的或者不好用的就在原有类的基础上进行封装扩展，例如 Kotlin 给 File 类写了扩展函数，这跟 Groovy 的扩展 API 的思想是一样的。

15.1.1 终端 IO 操作

Java 超长的输出语句 System.out.println() 居然延续到了现在！同样的工作在 C++ 里面只需要简单的 cout<< 就可以完成。当然，如果需要的话，我们可以在工程中直接封装 System.out.println() 为简单的打印方法。

在 Kotlin 里面很简单，只需要使用 println 或者 print 这两个全局函数即可，不再需要冗长的前缀。当然如果我们很怀旧，就是想用 System.out.println()，Kotlin 依然支持直接这么使用（与 Java 无缝互操作）：

```
>>> System.out.println("K")
```

```
K
>>> println("K")
K
```

这里的 println 函数 Kotlin 实现如下：

```
@kotlin.internal.InlineOnly
public inline fun println(message: Any?) {
    System.out.println(message)
}
```

当然，Kotlin 也只是在 System.out.println() 的基础上进行了封装。

从终端读取数据也很简单，最基本的方法就是全局函数 readLine，它直接从终端读取一行作为字符串。如果需要更进一步的处理，可以使用 Kotlin 提供的各种字符串处理函数来处理和转换字符串。

Kotlin 的封装终端 IO 的类在 stdlib/src/kotlin/io/Console.kt 源文件中。

15.1.2 读文件

Kotlin 为 java.io.File 提供了大量好用的扩展函数，这些扩展函数主要在下面三个源文件中：

- kotlin/io/files/FileTreeWalk.kt
- kotlin/io/files/Utils.kt
- kotlin/io/FileReadWrite.kt

同时，Kotlin 也针对 InputStream、OutputStream 和 Reader 等都做了简单的扩展。它们主要在下面的两个源文件中：

- kotlin/io/IOStreams.kt
- kotlin/io/ReadWrite.kt

Koltin 的序列化直接采用 Java 的序列化类的类型别名，如下所示：

```
internal typealias Serializable = java.io.Serializable
```

下面我们来简单介绍一下 Kotlin 文件读写操作。

1. 读取文件全部内容

如果简单读取一个文件，可以使用 readText() 方法，它直接返回整个文件内容。代码示例如下：

```
/**
 * 获取文件全部内容字符串
 * @param filename
 */
fun getFileContent(filename: String): String {
    val f = File(filename)
```

```
    return f.readText(Charset.forName("UTF-8"))
}
```

我们直接使用 File 对象来调用 readText 函数即可获得该文件的全部内容，readText 函数返回一个字符串。如果指定字符编码，可以通过传入参数 Charset 来指定，默认是 UTF-8 编码。

如果我们想要获得文件每行的内容，可以简单通过 split("\n") 来获得一个每行内容的数组。

2. 获取文件每行的内容

我们也可以直接调用 Kotlin 封装好的 readLines 函数，获得文件每行的内容。readLines 函数返回一个持有每行内容的 List：

```
/**
 * 获取文件每一行内容，存入 List 中
 * @param filename
 */
fun getFileLines(filename: String): List<String> {
    return File(filename).readLines(Charset.forName("UTF-8"))
}
```

3. 直接操作字节数组

如果希望直接操作文件的字节数组，可以使用 readBytes()。如果想使用传统的 Java 方式，在 Kotlin 中你也可以像 Groovy 一样自如使用：

```
// 读取为 bytes 数组
val bytes: ByteArray = f.readBytes()
println(bytes.joinToString(separator = " "))

// 直接像 Java 中的那样处理 Reader 或 InputStream
val reader: Reader = f.reader()
val inputStream: InputStream = f.inputStream()
val bufferedReader: BufferedReader = f.bufferedReader()
```

15.1.3　写文件

和读文件类似，写入文件也很简单。可以写入字符串，也可以写入字节流，还可以直接使用 Java 的 Writer 或者 OutputStream。

1. 覆盖写文件

把一段内容 text 写入文件中：

```
fun writeFile(text: String, destFile: String) {
    val f = File(destFile)
    if (!f.exists()) {
        f.createNewFile()
```

```
    }
    f.writeText(text, Charset.defaultCharset())
}
```

上面的代码每次运行时，该文件原来有的内容会被覆盖掉。

2. 末尾追加写文件

下面的代码每次在文件中默认追加写入 text 内容：

```
fun appendFile(text: String, destFile: String) {
    val f = File(destFile)
    if (!f.exists()) {
        f.createNewFile()
    }
    f.appendText(text, Charset.defaultCharset())
}
```

15.1.4 遍历文件树

和 Groovy 一样，在 Kotlin 中遍历文件树也很方便。遍历文件树需要调用扩展方法 walk()，它会返回一个 FileTreeWalk 对象，有一些方法用于设置遍历方向和深度，详情参见 FileTreeWalk API 文档说明。



提示 FileTreeWalk API 文档链接为 https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/-file-tree-walk/。

下面的例子遍历了指定文件夹下的所有文件：

```
fun traverseFileTree(filename: String) {
    val f = File(filename)
    val fileTreeWalk = f.walk()
    fileTreeWalk.iterator().forEach { println(it.absolutePath) }
}
```

测试代码：

```
@Test fun testTraverseFileTree() {
    KFileUtil.traverseFileTree(".")
}
```

运行上面的测试代码，将输出当前目录下的所有子目录及其文件。

我们还可以遍历当前文件下的所有子目录文件，存入一个 Iterator 中：

```
fun getFileIterator(filename: String): Iterator<File> {
    val f = File(filename)
    val fileTreeWalk = f.walk()
    return fileTreeWalk.iterator()
}
```

我们遍历当前文件下的所有子目录文件，还可以根据条件过滤，并把结果存入一个 Sequence 中：

```
fun getFileSequenceBy(filename: String, p: (File) -> Boolean): Sequence<File> {
    val f = File(filename)
    return f.walk().filter(p)
}
```

测试代码：

```
@Test fun testGetFileSequenceBy() {
    val fileSequence1 = KFileUtil.getFileSequenceBy(".", {
        it.isDirectory
    })
    fileSequence1.forEach { println("fileSequence1: ${it.absoluteFile} ") }

    val fileSequence2 = KFileUtil.getFileSequenceBy(".", {
        it.isFile
    })
    fileSequence2.forEach { println("fileSequence2: ${it.absoluteFile} ") }

    val fileSequence3 = KFileUtil.getFileSequenceBy(".", {
        it.extension == "kt"
    })
    fileSequence3.forEach { println("fileSequence3: ${it.absoluteFile} ") }
}
```

在工程中运行上面的测试代码，它将会有类似下面的输出：

```
...
...

fileSequence3: /Users/jack/kotlin/chapter15_file_io/./src/main/kotlin/com/
    easy/kotlin/fileio/KFileUtil.kt
fileSequence3: /Users/jack/kotlin/chapter15_file_io/./src/main/kotlin/com/
    easy/kotlin/fileio/KNetUtil.kt
fileSequence3: /Users/jack/kotlin/chapter15_file_io/./src/main/kotlin/com/
    easy/kotlin/fileio/KShellUtil.kt
fileSequence3: /Users/jack/kotlin/chapter15_file_io/./src/test/kotlin/com/
    easy/kotlin/fileio/KFileUtilTest.kt
```

15.1.5　网络 IO 操作

Kotlin 为 java.net.URL 增加了两个扩展方法：readBytes 和 readText。使用这两个方法配合正则表达式可以很方便地实现网络爬虫的功能。下面我们简单写几个函数实例。

根据 url 获取该 url 的响应 HTML 函数：

```
fun getUrlContent(url: String): String {
    return URL(url).readText(Charset.defaultCharset())
}
```

根据 url 获取该 url 的响应比特数组函数：

```
fun getUrlBytes(url: String): ByteArray {
    return URL(url).readBytes()
}
```

把 url 响应字节数组写入文件：

```
fun writeUrlBytesTo(filename: String, url: String) {
    val bytes = URL(url).readBytes()
    File(filename).writeBytes(bytes)
}
```

下面这个例子简单地获取了百度首页的源代码：

```
getUrlContent("https://www.baidu.com")
```

下面这个例子根据 url 来获取一张图片的比特流，然后调用 readBytes() 方法读取到字节流并写入文件：

```
writeUrlBytesTo(" 图片 .jpg", "http://n.sinaimg.cn/default/4_img/uplaod/3933d981/
    20170622/2fIE-fyhfxph6601959.jpg")
```

在项目相应文件夹下可以看到下载好的“图片 .jpg”。

15.1.6 kotlin.io 标准库

Kotlin 的 io 库主要是扩展 Java 的 io 库。下面我们简单举几个例子。

1. appendBytes

追加字节数组到该文件中。

方法签名：

```
fun File.appendBytes(array: ByteArray)
```

2. appendText

追加文本到该文件中。

方法签名：

```
fun File.appendText(
    text: String,
    charset: Charset = Charsets.UTF_8)
```

3. bufferedReader

获取该文件的 BufferedReader。

方法签名：

```
fun File.bufferedReader(
    charset: Charset = Charsets.UTF_8,
```

```
    bufferSize: Int = DEFAULT_BUFFER_SIZE
): BufferedReader
```

4. bufferedWriter

获取该文件的 BufferedWriter。

方法签名：

```
fun File.bufferedWriter(
    charset: Charset = Charsets.UTF_8,
    bufferSize: Int = DEFAULT_BUFFER_SIZE
): BufferedWriter
```

5. copyRecursively

复制该文件或者递归复制该目录及其所有子文件到指定路径，如果指定路径下的文件不存在，会自动创建。

方法签名：

```
fun File.copyRecursively(
    target: File,
    overwrite: Boolean = false, //是否覆盖。true: 覆盖之前先删除原来的文件
    onError: (File, IOException) -> OnErrorAction = { _, exception -> throw exception }
): Boolean
```



提示　Kotlin 对 File 的扩展函数 API 文档见 https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/java.io.-file/index.html。关于 kotlin.io 的 API 文档见 https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/index.html。

15.1.7　执行 Shell 命令行

Groovy 的文件 IO 操作非常好用，例如：

```
package com.easy.kotlin

import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4)
class ShellExecuteDemoTest {
    @Test
    def void testShellExecute() {
        def p = "ls -R".execute()
        def output = p.inputStream.text
        println(output)
        def fname = "我图.url"
        def f = new File(fname)
```

```
        def lines = f.readLines()
        lines.forEach({
            println(it)
        })
        println(f.text)
    }
}
```

Kotlin 中的文件 IO、网络 IO 操作跟 Groovy 一样简单。

另外，从上面的代码中我们看到使用 Groovy 执行终端命令非常简单：

```
def p = "ls -R".execute()
def output = p.inputStream.text
```

在 Kotlin 中，目前还没有对 String 类和 Process 扩展的函数。其实扩展这样的函数非常简单，我们完全可以自己扩展。

首先，我们来扩展 String 的 execute() 函数：

```
fun String.execute(): Process {
    val runtime = Runtime.getRuntime()
    return runtime.exec(this)
}
```

然后，给 Process 类扩展一个 text 函数：

```
fun Process.text(): String {
    var output = ""
    //输出 Shell 执行的结果
    val inputStream = this.inputStream
    val isr = InputStreamReader(inputStream)
    val reader = BufferedReader(isr)
    var line: String? = ""
    while (line != null) {
        line = reader.readLine()
        output += line + "\n"
    }
    return output
}
```

完成了上面两个简单的扩展函数之后，就可以在下面的测试代码中像 Groovy 一样执行终端命令了：

```
val p = "ls -al".execute()

val exitCode = p.waitFor()
val text = p.text()

println(exitCode)
println(text)
```

实际上，通过之前的很多实例学习，我们可以看出 Kotlin 的扩展函数相当实用。Kotlin 语言本身的 API 也大量使用了扩展功能。

15.2　正则表达式

我们在 Kotlin 中除了仍然可以使用 Java 中的 Pattern、Matcher 等类之外，Kotlin 还提供了一个正则表达式类 kotlin/text/regex/Regex.kt，可通过 Regex 的构造函数来创建一个正则表达式。下面详细介绍

15.2.1　构造 Regex 表达式

要构造 Regex 表达式，有两个方法：

- **使用 Regex 构造函数**。我们可以直接使用 Regex 类的构造函数来创建一个正则对象：

```
>>> val r1 = Regex("[a-z]+")
>>> val r2 = Regex("[a-z]+", RegexOption.IGNORE_CASE)
```

其中的匹配选项 RegexOption 直接使用 Java 类 Pattern 中的正则匹配选项。

- **使用 String 的 toRegex 扩展函数**。也可以直接使用 String 类的扩展函数把一个 String 转换成 Regex 对象：

```
>>> val r3 = "[A-Z]+".toRegex()
```

15.2.2　Regex 类中的函数

Regex 里面提供了丰富的简单而实用的函数，如表 15-1 所示。

表 15-1　Regex 函数及功能说明

函数名称	功能说明
`matches(input: CharSequence): Boolean`	输入字符串全部匹配
`containsMatchIn(input: CharSequence): Boolean`	输入字符串至少有一个匹配
`matchEntire(input: CharSequence): MatchResult?`	输入字符串全部匹配，返回一个匹配结果对象
`replace(input: CharSequence, replacement: String): String`	把输入字符串中匹配的部分替换成 replacement 的内容
`replace(input: CharSequence, transform: (MatchResult) -> CharSequence): String`	把输入字符串中匹配到的值，用函数 transform 映射之后的新值替换
`find(input: CharSequence, startIndex: Int = 0): MatchResult?`	返回输入字符串中第一个匹配值
`findAll(input: CharSequence, startIndex: Int = 0): Sequence`	返回输入字符串中所有匹配值 MatchResult 的序列

下面我们分别给出 Regex 类中的函数的简单实例。

1. matches

若输入字符串全部匹配，正则表达式返回 true，否则返回 false：

```
>>> val r1 = Regex("[a-z]+")
>>> r1.matches("ABCzxc")
false
>>>

>>> val r2 = Regex("[a-z]+", RegexOption.IGNORE_CASE)
>>> r2.matches("ABCzxc")
true

>>> val r3 = "[A-Z]+".toRegex()
>>> r3.matches("GGMM")
true
```

2. containsMatchIn

输入字符串中至少有一个匹配就返回 true，没有一个匹配就返回 false：

```
>>> val re = Regex("[0-9]+")
>>> re.containsMatchIn("012Abc")
true
>>> re.containsMatchIn("Abc")
false
```

3. matchEntire

若输入字符串全部匹配，正则表达式返回一个 MatcherMatchResult 对象，否则返回 null：

```
>>> val re = Regex("[0-9]+")
>>> re.matchEntire("1234567890")
kotlin.text.MatcherMatchResult@34d713a2
>>> re.matchEntire("1234567890!")
null
```

我们可以访问 MatcherMatchResult 的 value 来获得匹配的值：

```
>>> re.matchEntire("1234567890")?.value
1234567890
```

由于 matchEntire 函数的返回是 MatchResult? 可空对象，所以这里我们使用了安全调用符号 ?.。

4. replace(input: CharSequence, replacement: String): String

把输入字符串中匹配的部分替换成 replacement 的内容：

```
>>> val re = Regex("[0-9]+")
>>> re.replace("12345XYZ","abcd")
abcdXYZ
```

可以看到，“12345XYZ”中的 12345 是匹配正则表达式 [0–9]+ 的内容，它被替换成了 abcd。

5. replace(input: CharSequence, transform: (MatchResult) –> CharSequence): String

把输入字符串中匹配到的值，用函数 transform 映射之后的新值替换：

```
>>> val re = Regex("[0-9]+")
>>> re.replace("9XYZ8", { (it.value.toInt() * it.value.toInt()).toString() })
81XYZ64
```

可以看到，9XYZ8 中数字 9 和 8 是匹配正则表达式 [0–9]+ 的内容，它们分别被 transform 函数映射 (it.value.toInt() * it.value.toInt()).toString() 的新值 81 和 64 替换。

6. find

返回输入字符串中第一个匹配的 MatcherMatchResult 对象：

```
>>> val re = Regex("[0-9]+")
>>> re.find("123XYZ987abcd7777")
kotlin.text.MatcherMatchResult@4d4436d0
>>> re.find("123XYZ987abcd7777")?.value
123
```

7. findAll

返回输入字符串中所有匹配值的 MatchResult 序列：

```
>>> val re = Regex("[0-9]+")
>>> re.findAll("123XYZ987abcd7777")
kotlin.sequences.GeneratorSequence@f245bdd
```

我们可以通过 forEach 循环遍历所有匹配的值：

```
>>> re.findAll("123XYZ987abcd7777").forEach{println(it.value)}
123
987
7777
```

15.2.3　使用 Java 正则表达式类

除了上面 Kotlin 提供的函数之外，我们在 Kotlin 中仍然可以使用 Java 的正则表达式的 API：

```
val re = Regex("[0-9]+")
val p = re.toPattern()
val m = p.matcher("888ABC999")
while (m.find()) {
    val d = m.group()
    println(d)
}
```

上面的代码运行输出：

```
888
999
```

15.3 Kotlin 的多线程

Kotlin 中没有 synchronized 关键字，也没有 volatile 关键字。

Kotlin 的 Any 类似于 Java 的 Object，但是没有 wait()、notify() 和 notifyAll() 方法。

那么并发如何在 Kotlin 中工作呢？放心，Kotlin 既然是站在 Java 的肩膀上，当然少不了对多线程编程的支持。Kotlin 通过封装 Java 中的线程类，简化了编码。同时可以使用一些特定的注解，直接使用 Java 中的同步关键字等。下面我们简单介绍一下使用 Kotlin 进行多线程编程的相关内容。

15.3.1 创建线程

我们在 Java 中通常有两种方法创建线程：

- 扩展 Thread 类。
- 实例化它并通过构造函数传递一个 Runnable。

我们可以很容易地在 Kotlin 中使用 Java 类，这两个方式都可以使用。

1. 使用对象表达式创建

示例代码如下：

```
object : Thread() {
    override fun run() {
        Thread.sleep(3000)
        println("A 使用 Thread 对象表达式：${Thread.currentThread()}")
    }
}.start()
```

此代码使用 Kotlin 的对象表达式创建一个匿名类并覆盖 run() 方法。

2. 使用 Lambda 表达式

下面是如何将一个 Runnable 传递给一个新创建的 Thread 实例：

```
Thread({
    Thread.sleep(2000)
    println("B 使用 Lambda 表达式：${Thread.currentThread()}")
}).start()
```

我们在这里看不到 Runnable，在 Kotlin 中可以很方便地直接使用上面的 Lambda 表达式来表达。

还有更简单的方法吗？且看下文。

3. 使用 Kotlin 封装的 thread 函数

例如，我们写了下面一段线程的代码：

```
val t = Thread({
    Thread.sleep(2000)
    println("C 使用 Lambda 表达式 :${Thread.currentThread()}")
})
t.isDaemon = false
t.name = "CThread"
t.priority = 3
t.start()
```

后面的四行可以说是样板化的代码。在 Kotlin 中把这样的操作封装简化了：

```
thread(start = true, isDaemon = false, name = "DThread", priority = 3) {
    Thread.sleep(1000)
    println("D 使用 Kotlin 封装的函数 thread(): ${Thread.currentThread()}")
}
```

这样的代码显得更加精简整洁。事实上，thread() 函数就是对我们编程实践中经常用到的样板化的代码进行了抽象封装，它的实现如下：

```
public fun thread(start: Boolean = true, isDaemon: Boolean = false,
    contextClassLoader: ClassLoader? = null, name: String? = null,
    priority: Int = -1, block: () -> Unit): Thread {
    val thread = object : Thread() {
        public override fun run() {
            block()
        }
    }
    if (isDaemon)
        thread.isDaemon = true
    if (priority > 0)
        thread.priority = priority
    if (name != null)
        thread.name = name
    if (contextClassLoader != null)
        thread.contextClassLoader = contextClassLoader
    if (start)
        thread.start()
    return thread
}
```

这只是一个非常方便的包装函数，简单实用。从上面的例子我们可以看出，Kotlin 通过扩展 Java 的线程 API 简化了样板代码。

15.3.2　同步方法和块

synchronized 不是 Kotlin 中的关键字，它被替换为 @Synchronized 注解。Kotlin 中同步

方法的声明如下所示：

```
@Synchronized fun appendFile(text: String, destFile: String) {
    val f = File(destFile)
    if (!f.exists()) {
        f.createNewFile()
    }
    f.appendText(text, Charset.defaultCharset())
}
```

@Synchronized 注解与 Java 中的 synchronized 具有相同的效果：它会将 JVM 方法标记为同步。对于同步块，我们使用 synchronized() 函数，它使用锁作为参数：

```
fun appendFileSync(text: String, destFile: String) {
    val f = File(destFile)
    if (!f.exists()) {
        f.createNewFile()
    }

    synchronized(this){
        f.appendText(text, Charset.defaultCharset())
    }
}
```

跟 Java 基本一样。

15.3.3 可变字段

同样，Kotlin 没有 volatile 关键字，但是有 @Volatile 注解：

```
@Volatile private var running = false
fun start() {
    running = true
    thread(start = true) {
        while (running) {
            println("Still running: ${Thread.currentThread()}")
        }
    }
}

fun stop() {
    running = false
    println("Stopped: ${Thread.currentThread()}")
}
```

@Volatile 注解和 Java 的 volatile 修饰符的作用是一样的。

当然，在 Kotlin 中我们有更好用的协程并发库。在工程实践中，我们可以根据实际情况自由选择。

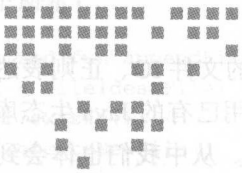
15.4　本章小结

Kotlin 是一门工程实践性很强的语言，从本章介绍的文件 IO、正则表达式以及多线程等内容中，我们可以领会到 Kotlin 的基本原则：充分使用已有的 Java 生态库，在此基础之上进行更加简单实用的扩展，大大提升程序员的生产力。从中我们也体会到了 Kotlin 编程中的极简理念——不断地抽象、封装、扩展，使之更加简单实用。

本章示例代码：https://github.com/EasyKotlin/chapter15_file_io。

另外，笔者综合了本章的内容，使用 SpringBoot + Kotlin 写了一个简单的图片爬虫 Web 应用，感兴趣的读者可参考源码：https://github.com/EasyKotlin/chatper15_net_io_img_crawler。

在下一章，也是我们的最后一章中，让我们脱离 JVM，直接使用 Kotlin Native 来开发一个可编译成机器码运行的 Kotlin 应用程序。



Chapter 16 第 16 章

使用 Kotlin Native

不得不说 JetBrains 是一家务实的公司，各种 IDE 让人赞不绝口，用起来也是相当顺手。同样，诞生自 JetBrains 的 Kotlin 也是一门务实的编程语言，Kotlin 以工程实用性为导向，充分借鉴了 Java、Scala、Groovy、C#、Gosu、JavaScript、Swift 等语言的精华，让我们写起代码来相当优雅却又不失工程质量与效率。Kotlin Native 能把 Kotlin 代码直接编译成机器码，也就是站在了与 C/C++、Go 和 Rust 同一个层次上，为这个领域又添了一位竞争对手。

在前面的所有章节中，我们使用的 Kotlin 都是基于 JVM 的运行环境。本章将从 JVM 的运行环境中离开，走向直接编译生成原生机器码的系统编程的生态系统：Kotlin Native。

16.1 Kotlin Native 简介

Kotlin Native 利用 LLVM 来编译到机器码。Kotlin Native 主要基于 LLVM 后端编译器来生成本地机器码。

Kotlin Native 的设计初衷是为了支持在非 JVM 虚拟机平台环境的编程，如 iOS、嵌入式平台等，同时支持与 C 互操作。

1. LLVM

LLVM（Low Level Virtual Machine）定位是一个虚拟机，并且是比较底层的虚拟机。LLVM 是构架编译器的框架系统，用 C++ 编写而成，用于优化任意程序（用任何程序语言编写）的编译时间（compile-time）、链接时间（link-time）、运行时间（run-time）以及空闲时间（idle-time），对开发者保持开放，并兼容已有脚本。

LLVM 的出现是为了解决编译器代码重用的问题，LLVM 一上来就站在比较高的高度，

制定了 LLVM IR 这一中间代码表示语言。LLVM IR 充分考虑了各种应用场景，例如在 IDE 中调用 LLVM 进行实时的代码语法检查，对静态语言、动态语言的编译、优化等。

2. 支持平台

Kotlin Native 现在已支持的平台参见表 16-1。

表 16-1　Kotlin Native 现在支持的平台

平台名称	target 配置	平台名称	target 配置
Linux	linux	Android arm64	android_arm64
Mac OS	macbook	iOS	iphone
Windows	mingw	Raspberry Pi	raspberrypi
Android arm32	android_arm32		

这意味着我们可以在这些平台上愉快地开始体验了！目前 Kotlin Native 已经发布的最新预版本是 v0.3。

3. 解释型语言与编译型语言

编译型语言是指在程序执行之前有一个单独的编译过程，将程序翻译成机器语言，以后执行这个程序的时候，就不用再进行翻译了。例如，C/C++ 等都是编译型语言。

解释型语言是指在运行的时候将程序翻译成机器语言，所以运行速度相对于编译型语言要慢。例如，Java、C# 等都是解释型语言。

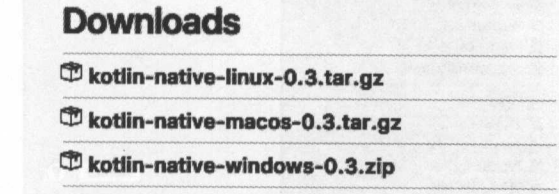
虽然 Java 程序在运行之前也有一个编译过程，但是并不是将程序编译成机器语言，而是将它编译成字节码（可以理解为一个中间语言）。在运行的时候，由 JVM 将字节码再翻译成机器语言。

针对之前讲的基于 JVM 的 Kotlin 标准库部分，我们可以把 Kotlin 理解为解释型语言。但是通过本章 Kotlin Native 的学习我们会发现，也可以说 Kotlin 是一种编译型语言。

16.2　快速开始 Hello World

1. 运行环境准备

我们直接去 GitHub 上面下载 kotlin-native 编译器的软件包，地址是 https://github.com/JetBrains/kotlin-native/releases，软件包如下所示：



下载解压之后，我们可以看到 Kotlin Native 编译器 konan 的目录如下：

```
-rw-r--r--@  1 jack  staff   6828  6 20 22:47 GRADLE_PLUGIN.md
-rw-r--r--@  1 jack  staff  16286  6 20 22:47 INTEROP.md
-rw-r--r--@  1 jack  staff   1957  6 21 01:03 README.md
-rw-r--r--@  1 jack  staff   4606  6 20 22:47 RELEASE_NOTES.md
drwxr-xr-x@  8 jack  staff    272  6 20 23:04 bin
drwxr-xr-x   6 jack  staff    204  7 28 17:08 dependencies
drwxr-xr-x@  3 jack  staff    102  6 20 23:01 klib
drwxr-xr-x@  5 jack  staff    170  5 12 00:02 konan
drwxr-xr-x@  4 jack  staff    136  5 12 00:02 lib
drwxr-xr-x@ 22 jack  staff    748  6 22 19:04 samples
```

关于这个目录里面的内容我们在后面小节中介绍。

另外，我们也可以自己下载源码编译 kotlin-native。

首先从 GitHub 下载 Kotlin Native 源码，执行如下命令：

```
cd kotlin-native
```

执行下载依赖包命令：

```
./gradlew dependencies:update
```

编译：

```
./gradlew dist
```

至此，我们就得到了 Kotlin 的 Native 编译器，它们位于目录 ./dist/bin。

2. 新建 Gradle 项目

在本小节中，我们先使用 IDEA 来创建一个普通的 Gradle 项目。

1）打开 File → New → Project，如图 16-1 所示。

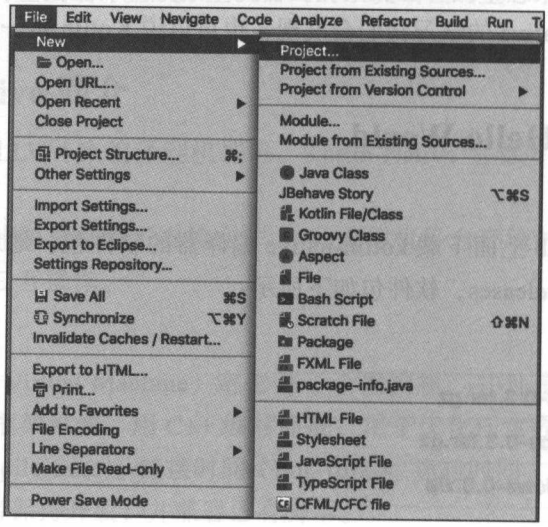


图 16-1 新建工程

2）可直接在左侧栏中选择 Gradle，点击 Next，如图 16-2 所示。

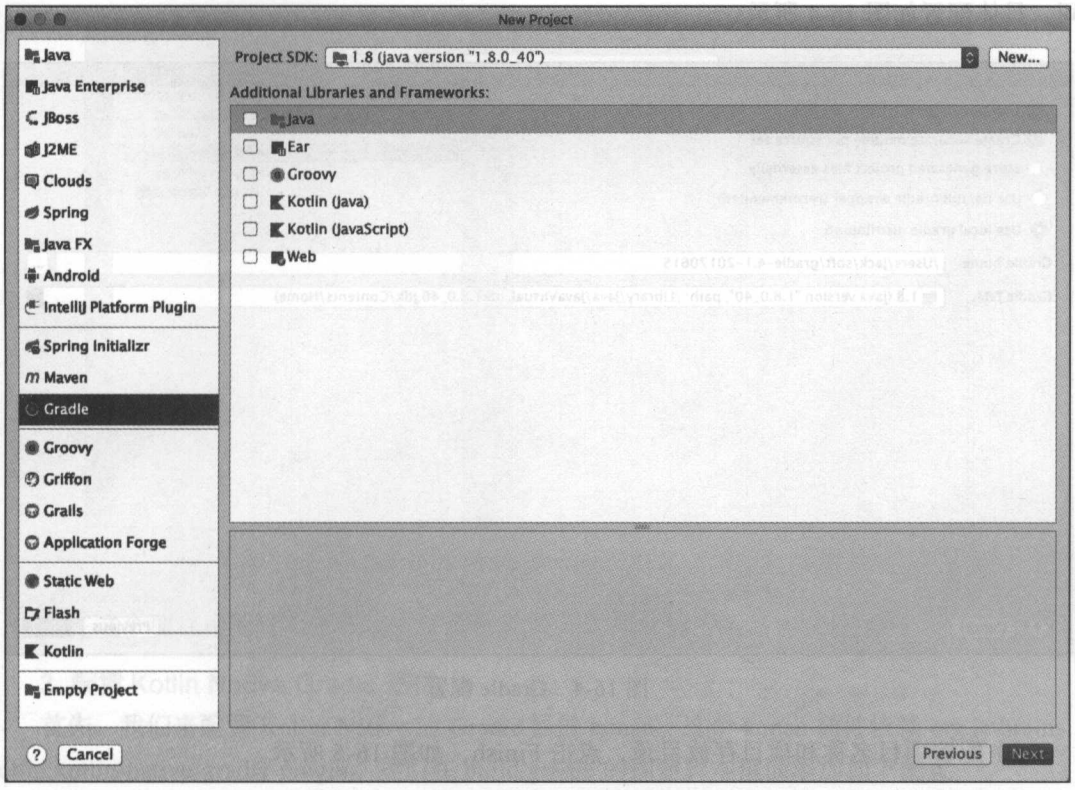


图 16-2　选择 Gradle 项目类型

3）设置项目的 GroupId、ArtifactId、Version 信息，如图 16-3 所示。

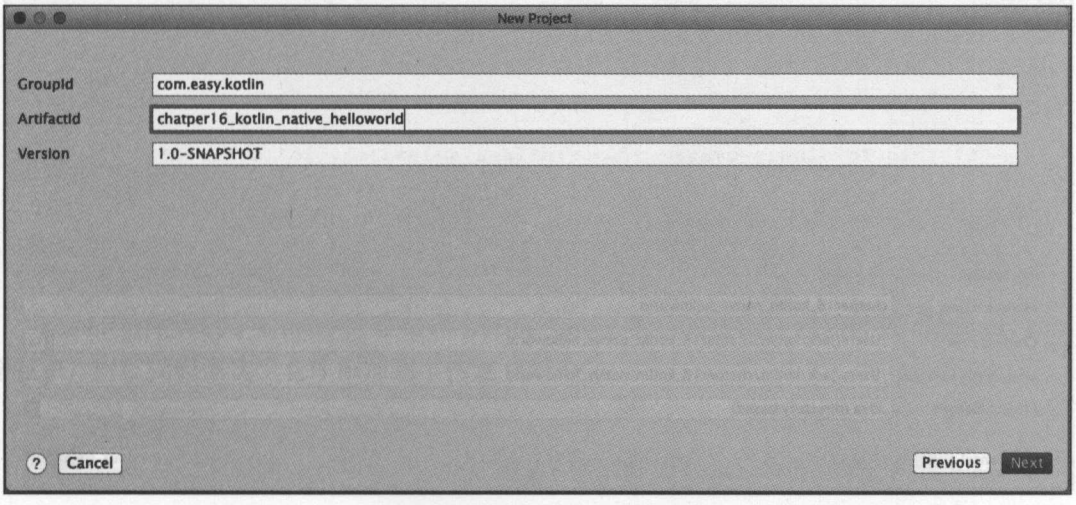


图 16-3　设置项目信息

4）配置 Gradle 项目的基本设置。可直接选择本地的 Gradle 环境目录，省去下载的时间，具体配置如图 16-4 所示。

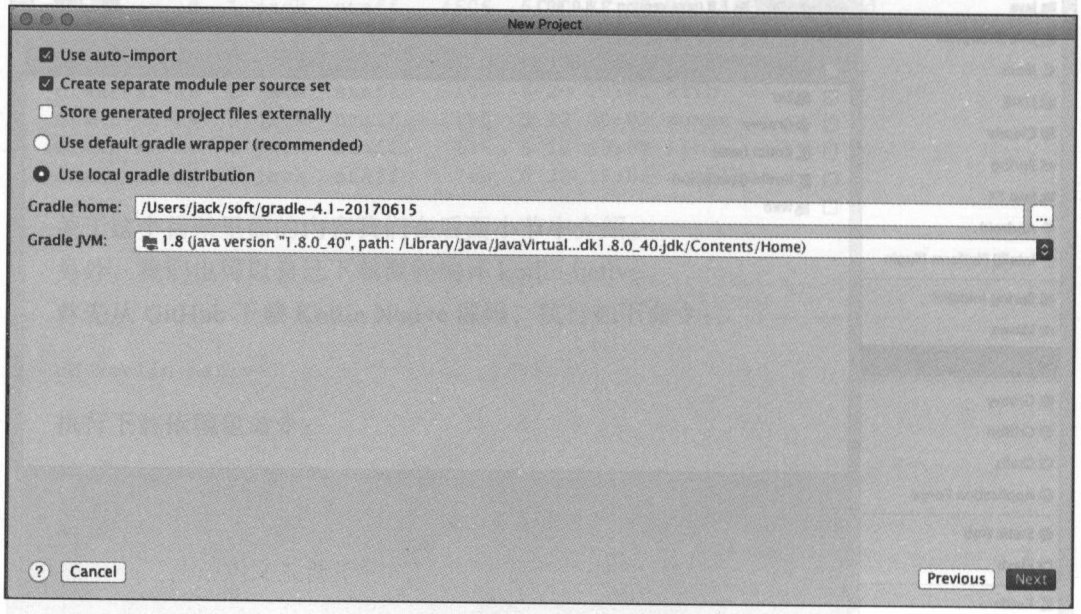


图 16-4 Gradle 配置

5）配置项目名称和项目存放目录，点击 Finish，如图 16-5 所示。

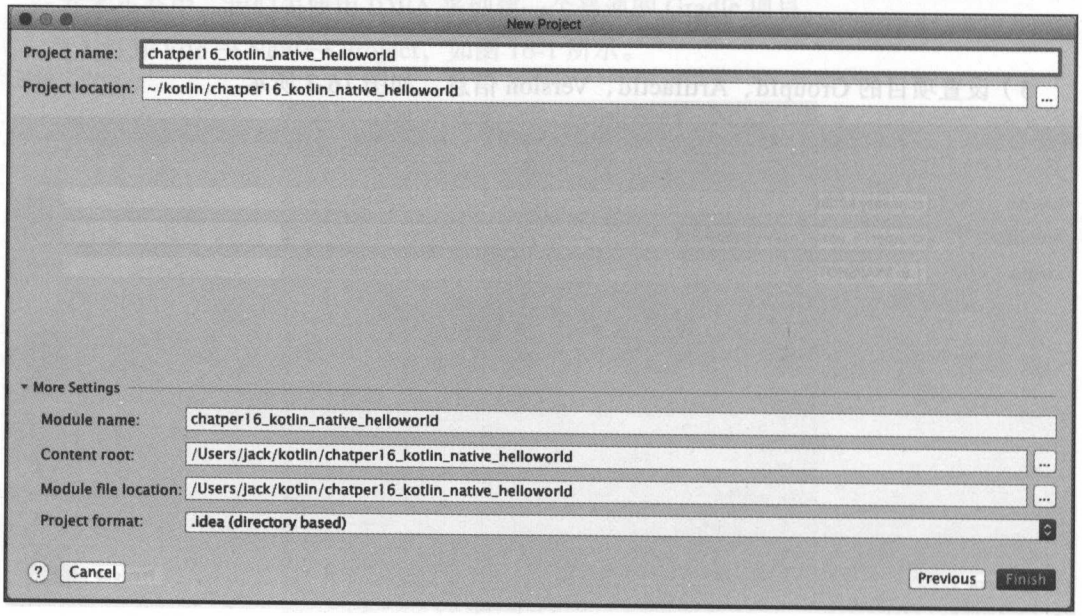


图 16-5 配置项目名称目录

6）等待 IDEA 创建完毕，我们将得到的 Gradle 项目，如图 16-6 所示。

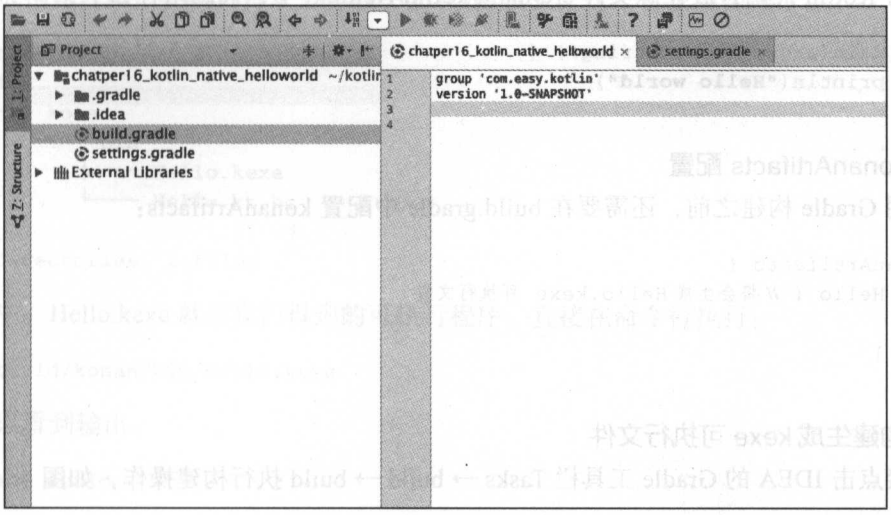


图 16-6　标准 Gradle 空项目

现在这个项目里面什么都没有。下面我们就开始原始的手工新建文件编码。

3. 配置 Kotlin Native Gradle 插件

首先，我们来配置 Kotlin Native 的 Gradle 插件 konan。这个 konan 插件依赖 org.jetbrains.kotlin:kotlin-native-gradle-plugin。我们在 build.gradle 的构建脚本里面添加如下配置：

```
buildscript {
    repositories {
        mavenCentral()
        maven {
            url "https://dl.bintray.com/jetbrains/kotlin-native-dependencies"
        }
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-native-gradle-plugin:0.3"
    }
}

apply plugin: 'konan'
```

konan 插件集成了 Gradle 构建编译 Kotlin Native 项目的功能，该插件在 dl.bintray.com 仓库中，所以我们把这个仓库添加到了构建仓库中。

4. 配置 Kotlin Native 运行环境

我们在根目录下新建 gradle.properties 文件，里面添加 konan.home 的配置：

```
konan.home=/Users/jack/soft/kotlin-native-macos-0.3
```

5. 编写 Hello.kt

编写 Kotlin 代码，放在源文件 src/main/kotlin/Hello.kt 中，代码内容如下：

```
fun main(args: Array<String>) {
    println("Hello world")
}
```

6. konanArtifacts 配置

使用 Gradle 构建之前，还需要在 build.gradle 中配置 konanArtifacts：

```
konanArtifacts {
    Hello { //将会生成 Hello.kexe 可执行文件

    }
}
```

7. 构建生成 kexe 可执行文件

直接点击 IDEA 的 Gradle 工具栏 Tasks → build → build 执行构建操作，如图 16-7 所示。

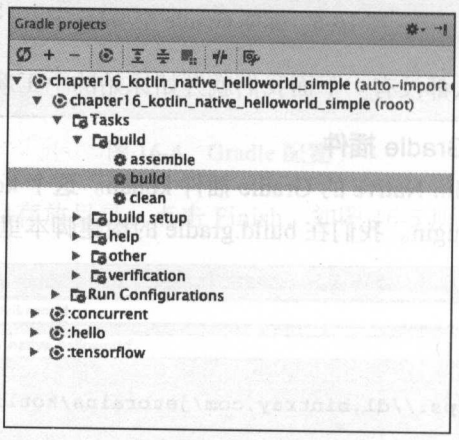


图 16-7 构建生成 kexe 可执行文件

此刻我们可以看到控制台输出：

```
22:49:13: Executing external task 'build'...
:assemble UP-TO-DATE
:check UP-TO-DATE
:downloadKonanCompiler
:compileKonanHello UP-TO-DATE
:compileKonan UP-TO-DATE
:build UP-TO-DATE

BUILD SUCCESSFUL in 0s
2 actionable tasks: 1 executed, 1 up-to-date
22:49:13: External task execution finished 'build'.
```

8. 运行测试

完成构建之后，在项目目录下生成一个 build 目录，这个目录结构如下：

```
$ tree build
build
└── konan
    └── bin
        ├── Hello.kexe
        └── Hello.kt.bc

2 directories, 2 files
```

其中，Hello.kexe 就是我们得到的可执行程序。直接在命令行执行：

```
$ build/konan/bin/Hello.kexe
```

可以看到输出：

```
Hello world
```

这就是我们通过使用 Kotlin Native 编译器，直接把 Kotlin 代码编译成机器码可执行程序的简单过程。

本节示例工程源码：https://github.com/EasyKotlin/chapter16_kotlin_native_helloworld_simple

---

16.3　Kotlin Native 与 C 语言互操作

本节介绍如何通过使用 Kotlin Native 来调用 C 语言代码接口。新建标准 Gradle 的过程与上节操作一样，此处我们省去此过程，直接从源代码目录结构设计开始。

1. 源代码目录

首先在根目录下面新建 src 目录，用来存放源代码。在 src 下面新建 c 目录存放 C 代码，新建 kotlin 目录存放 Kotlin 代码。我们的源代码组织结构设计如下所示：

```
src
├── c
│   ├── cn_kotlinor.c
│   └── cn_kotlinor.h
└── kotlin
    └── main.kt
```

2. C 代码文件

C 头文件 cn_kotlinor.h 的声明如下：

```
#ifndef CN_KOTLINOR_H
```

```
#define CN_KOTLINOR_H
void printHello();
int factorial(int n);
int fib(int n);
#endif
```

我们简单声明了 3 个函数。

C 源代码文件 cn_kotlinor.c 内容如下：

```
#include "cn_kotlinor.h"
#include <stdio.h>

void printHello(){
    printf("[C]HelloWorld\n");
}

int factorial(int n){
    printf("[C]calc factorial: %d\n", n);
    if(n == 0) return 1;
    return n * factorial(n - 1);
}

int fib(int n){
    printf("[C]calc fibonacci: %d\n", n);
    if(n==1||n==2) return 1;
    return fib(n-1) + fib(n-2);
}
```

这就是我们熟悉的 C 语言代码。

3. Kotlin 代码文件

main.kt 文件内容如下：

```
import ckotlinor.*

fun main(args: Array<String>) {
    printHello()
    (1..7).map(::factorial).forEach(::println)
    (1..7).map(::fib).forEach(::println)
}
```

其中，import kotlinor.* 是 C 语言代码经过 clang 编译之后的 C 接口包路径，我们将在下面的 build.gradle 配置文件中的 konanInterop 中配置这个路径。

4. konan 插件配置

首先，我们在 build.gradle 里面添加构建脚本 buildscript 闭包：

```
buildscript {
    repositories {
        mavenCentral()
```

```
        maven {
            url "https://dl.bintray.com/jetbrains/kotlin-native-dependencies"
        }
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-native-gradle-plugin:0.3"
    }
}
```

这里我们添加了 Gradle 构建 Kotlin Native 工程的 DSL 插件 kotlin-native-gradle-plugin:0.3。这里的版本号对应我们下载的 konan 编译器的版本号，我们使用的是 v0.3，所以这里我们也使用 0.3 版本的插件。这个插件发布在 https://dl.bintray.com/jetbrains/kotlin-native-dependencies 仓库里，所以我们在 repositories 里面添加了这个仓库。

然后，我们应用插件 konan：

```
apply plugin: 'konan'
```

konan 就是用来编译 Kotlin 为 native 代码的插件。

5. konanInterop 互操作配置

konanInterop 主要用来配置 Kotlin 调用 C 的接口：

```
konanInterop {
    ckotlinor {
        defFile 'kotlinor.def' //interop 的配置文件
        includeDirs "src/c"    //C 头文件目录，可以传入多个
    }
}
```

konanInterop 的配置是由 konan 插件 API 中的 KonanInteropTask.kt 来处理的（这个类的源码在 https://github.com/JetBrains/kotlin-native/blob/master/tools/kotlin-native-gradle-plugin/src/main/kotlin/org/jetbrains/kotlin/gradle/plugin/KonanInteropTask.kt）。

这里我们声明的 ckotlinor 是插件中的 KonanInteropConfig 对象。我们在下面的 konanArtifacts 里面会引用这个 ckotlinor。

关于 konanInterop 的配置选项有：

```
konanInterop {
    pkgName {
        defFile <def-file>
        pkg <package with stubs>
        target <target: linux/macbook/iphone/iphone_sim>
        compilerOpts <Options for native stubs compilation>
        linkerOpts <Options for native stubs >
        headers <headers to process>
        includeDirs <directories where headers are located>
        linkFiles <files which will be linked with native stubs>
        dumpParameters <Option to print parameters of task before execution>
```

```
    }
    // TODO: add configuration for konan compiler
}
```

简要说明如表 16-2 所示。

表 16-2 konanInterop 的配置选项及功能说明

配置项	功能说明
defFile	互操作映射关系配置文件
pkg	C 头文件编译后映射为 Kotlin 的包名
target	编译目标平台：linux/macbook/iphone/iphone_sim 等
compilerOpts	编译选项
linkerOpts	链接选项
headers	要处理的头文件
includeDirs	包括的头文件目录
linkFiles	与 native stubs 链接的文件
dumpParameters	打印 Gradle 任务参数选项配置

其中，kotlinor.def 是 Kotlin Native 与 C 语言互操作的配置文件，我们在 kotlinor.def 里面配置 C 源码到 kotlin 的映射关系。这个 kotlinor.def 文件内容如下：

```
headers=cn_kotlinor.h
compilerOpts=-Isrc/c
```

同样的配置，如果我们写在 build.gradle 文件中的 konanInterop 配置里如下所示：

```
konanInterop {
    ckotlinor {
        // defFile 'kotlinor.def' // interop 的配置文件
        compilerOpts '-Isrc/c'
        headers 'src/c/cn_kotlinor.h' // interop 的配置文件
        includeDirs "src/c" // C 头文件存放目录，可以传入多个
    }
}
```

关于这个配置文件的解析原理可以参考 KonanPlugin.kt 文件的源码（https://github.com/JetBrains/kotlin-native/blob/master/tools/kotlin-native-gradle-plugin/src/main/kotlin/org/jetbrains/kotlin/gradle/plugin/KonanPlugin.kt）。

6. konanArtifacts 配置

在 konan 插件中，我们使用 konanArtifacts 来配置编译任务执行：

```
konanArtifacts {
    KotlinorApp { //①
        inputFiles fileTree("src/kotlin") //②
        useInterop 'ckotlinor' //③
        nativeLibrary fileTree('src/c/cn_kotlinor.bc') //④
```

```
        target 'macbook' // ⑤
    }
}
```

其中：

① KotlinorApp 名称在 build 之后会生成以这个名称命名的 KotlinorApp.kexe 可执行程序。

② inputFiles 配置的是 kotlin 代码目录，程序执行的入口 main 定义在这里。

③ useInterop 配置的是使用哪个互操作配置，我们使用的是前面的 konanInterop 里面的配置 ckotlinor。

④ nativeLibrary 配置的是本地库文件。关于 'src/c/cn_kotlinor.bc' 文件的编译生成我们在下面讲。

⑤ target 配置的是编译的目标平台，这里我们配置为 'macbook'。

关于 konanArtifacts 可选的配置如下所示：

```
konanArtifacts {

    artifactName1 {

        inputFiles "files" "to" "be" "compiled"

        outputDir "path/to/output/dir"

        library "path/to/library"
        library File("Library")

        nativeLibrary "path/to/library"
        nativeLibrary File("Library")

        noStdLib
        produce "library"|"program"|"bitcode"
        enableOptimization

        linkerOpts "linker" "args"
            target "target"

        languageVersion "version"
        apiVersion "version"

    }
        artifactName2 {

            extends artifactName1

            inputDir "someDir"
            outputDir "someDir"
        }

    }
```

konan 编译任务配置处理类是 KonanCompileTask.kt（https://github.com/JetBrains/kotlin-native/blob/master/tools/kotlin-native-gradle-plugin/src/main/kotlin/org/jetbrains/kotlin/gradle/plugin/KonanCompileTask.kt）。

7. 完整的 build.gradle 配置

完整的 build.gradle 配置文件内容如下：

```
group 'com.easy.kotlin'
version '1.0-SNAPSHOT'

buildscript {
    repositories {
        mavenCentral()
        maven {
            url "https://dl.bintray.com/jetbrains/kotlin-native-dependencies"
        }
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-native-gradle-plugin:0.3"
    }
}

apply plugin: 'konan' //konan 就是用来编译 Kotlin 为 native 代码的插件

konanInterop { //konanInterop 主要用来配置 Kotlin 调用 C 的接口
    ckotlinor {
        defFile 'kotlinor.def' //interop 的配置文件
        includeDirs "src/c" //C 头文件目录，可以传入多个
    }
}

konanArtifacts { //konanArtifacts 配置我们的项目
    KotlinorApp { //build 之后会生成 KotlinorApp.kexe 可执行程序
        inputFiles fileTree("src/kotlin") //kotlin 代码配置，项目入口 main 需要定义在这里
        useInterop 'ckotlinor' //使用前面的 konanInterop 里面的配置 kotlinor{ ... }
        nativeLibrary fileTree('src/c/cn_kotlinor.bc') //自己编译的 llvm 字节格式的依赖
        target 'macbook' //编译的目标平台
    }
}
```



关于 konan 插件的详细配置文档 Gradle DSL 见 https://github.com/JetBrains/kotlin-native/blob/master/GRADLE_PLUGIN.md。

8. 使用 clang 编译 C 代码

为了实用性，我们新建一个 shell 脚本 kclang.sh 来简化 clang 编译的命令行输入参数：

```
#!/usr/bin/env bash
clang -std=c99 -c $1 -o $2 -emit-llvm
```

我们把 kclang.sh 放到 C 代码目录下，然后直接使用脚本来编译：

```
kclang.sh cn_kotlinor.c cn_kotlinor.bc
```

将得到一个 cn_kotlinor.bc 库文件。

clang 是一个 C++ 编写、基于 LLVM、发布于 LLVM BSD 许可证下的 C/C++/Objective-C/Objective-C++ 编译器。它与 GNU C 语言规范几乎完全兼容。更多关于 clang 的内容可参考 http://clang.llvm.org/docs/index.html。

9. 配置 konan 编译器主目录

最后，在执行 Gradle 构建之前，我们还需要指定 konan 编译器主目录。我们在根目录下面新建 gradle.properties 这个属性配置文件，内容如下：

```
konan.home=/Users/jack/soft/kotlin-native-macos-0.3
```

10. 执行构建操作

我们直接在 IDEA 右侧的 Gradle 工具栏点击 Tasks → build → build 命令执行构建操作，如图 16-8 所示。

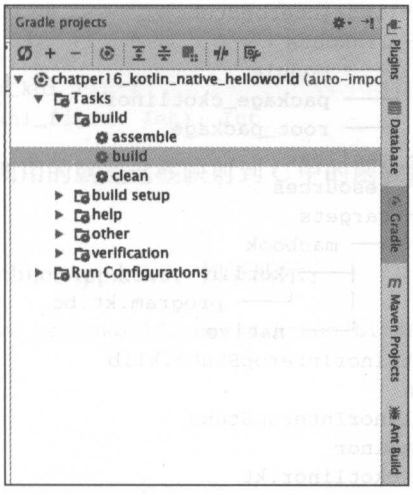


图 16-8　执行构建

会看到终端输出：

```
15:12:02: Executing external task 'build'...
:assemble UP-TO-DATE
```

```
:check UP-TO-DATE
:downloadKonanCompiler
:genKotlinerInteropStubs
:compileKotlinerInteropStubs
KtFile: kotliner.kt
:compileKonanKotliner
KtFile: main.kt
ld: warning: object file (/var/folders/q5/kvt7_nsd6ngdw5qry4d99xv00000gn/T/
    combined697750051437954502.o) was built for newer OSX version (10.12)
    than being linked (10.11)
:compileKonan
:build

BUILD SUCCESSFUL in 29s
4 actionable tasks: 4 executed
15:12:31: External task execution finished 'build'.
```

构建完成之后，会在 build/konan/bin/ 目录下面生成一个 KotlinorApp.kexe 可执行程序，它直接在 Mac OS 上运行，不再依赖 JVM 环境了。我们得到的完整构建输出目录树如下：

```
build
└── konan
    ├── bin
    │   ├── KotlinorApp.kexe
    │   └── KotlinorApp.kt.bc
    ├── interopCompiledStubs
    │   └── ckotlinorInteropStubs
    │       ├── ckotlinorInteropStubs
    │       │   ├── linkdata
    │       │   │   ├── module
    │       │   │   ├── package_ckotlinor
    │       │   │   └── root_package
    │       │   ├── manifest
    │       │   ├── resources
    │       │   └── targets
    │       │       └── macbook
    │       │           ├── kotlin
    │       │           │   └── program.kt.bc
    │       │           └── native
    │       └── ckotlinorInteropStubs.klib
    ├── interopStubs
    │   └── genCkotlinorInteropStubs
    │       └── ckotlinor
    │           └── ckotlinor.kt
    └── nativelibs
        └── genCkotlinorInteropStubs
            └── ckotlinorstubs.bc

16 directories, 10 files
```

在 ckotlinor.kt 中，可以看出 konan 编译器还为我们生成了 C 代码对应的 Kotlin 的接口：

```
@file:Suppress("UNUSED_EXPRESSION", "UNUSED_VARIABLE")
package ckotlinor

import konan.SymbolName
import kotlinx.cinterop.*

fun printHello(): Unit {
    val res = kni_printHello()
    return res
}

@SymbolName("ckotlinor_kni_printHello")
private external fun kni_printHello(): Unit

fun factorial(n: Int): Int {
    val _n = n
    val res = kni_factorial(_n)
    return res
}

@SymbolName("ckotlinor_kni_factorial")
private external fun kni_factorial(n: Int): Int

fun fib(n: Int): Int {
    val _n = n
    val res = kni_fib(_n)
    return res
}

@SymbolName("ckotlinor_kni_fib")
private external fun kni_fib(n: Int): Int
```

我们在 Kotlin 代码中，调用的就是这些映射到 C 中的函数接口。

11. 执行 kexe 应用程序

直接在命令行中执行 KotlinorApp.kexe，如下所示：

```
chatper16_kotlin_native_helloworld$ build/konan/bin/KotlinorApp.kexe
```

我们可以看到如下输出：

```
[C]HelloWorld
[C]calc factorial: 1
[C]calc factorial: 0
[C]calc factorial: 2
...
[C]calc factorial: 2
[C]calc factorial: 1
[C]calc factorial: 0
1
```

```
2
6
24
120
720
5040
[C]calc fibonacci: 1
[C]calc fibonacci: 2
[C]calc fibonacci: 3
...
[C]calc fibonacci: 3
[C]calc fibonacci: 2
[C]calc fibonacci: 1
1
1
2
3
5
8
13
```

至此，我们完成了一次简单的 Kotlin Native 与 C 语言互操作在系统级编程的体验之旅。完整工程源代码见本章结尾。

我们看到，Kotlin Native 仍然看重互操作性，它能高效地调用 C 函数，甚至还能从 C 头文件自动生成对应的 Kotlin 接口，发扬了 JetBrains 为开发者服务的良好传统！

但是，在体验的过程中我们也发现整个过程比较手工化，显得比较繁琐（例如手工新建各种配置文件、手工使用 clang 编译 C 代码等）。

不过，Kotlin Native 的 Gradle 插件用起来还是相当不错的。相信未来 IDEA 会对 Kotlin Native 开发进行智能的集成，以方便系统编程的开发者更好、更快地完成项目的配置以及开发编码工作。

16.4 Kotlin Native 编译器 konan 简介

本小节我们简单介绍一下 Kotlin Native 编译器 konan 的相关内容（主要以 Mac OS 平台示例）。

我们下载的 Kotlin Native 编译器 zip 压缩包解压后的二级目录如下：

```
$ tree -L 2
.
├── GRADLE_PLUGIN.md
├── INTEROP.md
├── README.md
├── RELEASE_NOTES.md
├── bin
```

```
|    ├── cinterop
|    ├── klib
|    ├── konanc
|    ├── kotlinc
|    ├── kotlinc-native
|    └── run_konan
├── dependencies
|    ├── clang-llvm-3.9.0-darwin-macos
|    ├── libffi-3.2.1-2-darwin-macos
|    └── target-sysroot-1-darwin-macos
├── klib
|    └── stdlib
├── konan
|    ├── konan.properties
|    ├── lib
|    └── nativelib
├── lib
|    ├── host
|    └── iphone
└── samples
    ├── README.md
    ...
```

下面主要针对这些目录模块中的功能作简单介绍。

1. bin 目录

bin 目录下面是执行命令行：

```
cinterop      klib      konanc      kotlinc      kotlinc-native  run_konan
```

run_konan 是真正的入口 shell，它的执行逻辑是：

```
TOOL_NAME="$1"
shift

if [ -z "$JAVACMD" -a -n "$JAVA_HOME" -a -x "$JAVA_HOME/bin/java" ]; then
    JAVACMD="$JAVA_HOME/bin/java"
else
    JAVACMD=java
fi
[ -n "$JAVACMD" ] || JAVACMD=java
...
java_opts=(-ea \
            -Xmx3G \
            "-Djava.library.path=${NATIVE_LIB}" \
            "-Dkonan.home=${KONAN_HOME}" \
           -Dfile.encoding=UTF-8)

KONAN_JAR="${KONAN_HOME}/konan/lib/backend.native.jar"
KOTLIN_JAR="${KONAN_HOME}/konan/lib/kotlin-compiler.jar"
```

```
STUB_GENERATOR_JAR="${KONAN_HOME}/konan/lib/StubGenerator.jar"
INTEROP_INDEXER_JAR="${KONAN_HOME}/konan/lib/Indexer.jar"
INTEROP_JAR="${KONAN_HOME}/konan/lib/Runtime.jar"
HELPERS_JAR="${KONAN_HOME}/konan/lib/helpers.jar"
KLIB_JAR="${KONAN_HOME}/konan/lib/klib.jar"
UTILITIES_JAR="${KONAN_HOME}/konan/lib/utilities.jar"
KONAN_CLASSPATH="$KOTLIN_JAR:$INTEROP_JAR:$STUB_GENERATOR_JAR:$INTEROP_INDEXER_
    JAR:$KONAN_JAR:$HELPERS_JAR:$KLIB_JAR:$UTILITIES_JAR"
TOOL_CLASS=org.jetbrains.kotlin.cli.utilities.MainKt

LIBCLANG_DISABLE_CRASH_RECOVERY=1 \
$TIMECMD "$JAVACMD" "${java_opts[@]}" "${java_args[@]}" -cp "$KONAN_CLASSPATH"
    "$TOOL_CLASS" "$TOOL_NAME" "${konan_args[@]}"
```

可以看出，Kotlin Native 编译器 konan 的运行环境还是在 JVM 上，但是它生成的机器码的可执行程序直接运行在对应的平台系统上（直接编译成机器语言）。

2. konan 目录

konan 目录是 Kotlin Native 编译器的核心实现部分，目录结构如下：

```
kotlin-native-macos-0.3$ tree konan
konan/
├── konan.properties
├── lib
│   ├── Indexer.jar
│   ├── Runtime.jar
│   ├── StubGenerator.jar
│   ├── backend.native.jar
│   ├── callbacks
│   │   └── shared
│   │       └── libcallbacks.dylib
│   ├── clangstubs
│   │   └── shared
│   │       └── libclangstubs.dylib
│   ├── helpers.jar
│   ├── klib.jar
│   ├── kotlin-compiler.jar
│   ├── protobuf-java-2.6.1.jar
│   └── utilities.jar
└── nativelib
    ├── libcallbacks.dylib
    ├── libclangstubs.dylib
    ├── libllvmstubs.dylib
    └── liborgjetbrainskotlinbackendkonanhashstubs.dylib

6 directories, 16 files
```

可以看到，在 run_konan 命令行 shell 中需依赖上面的这些 jar 包。上面的目录文件是 Mac OS 平台上的。

对应的 Linux 平台上的 konan 目录文件如下：

```
kotlin-native-linux-0.3$ tree konan
konan
├── konan.properties
├── lib
│   ├── Indexer.jar
│   ├── Runtime.jar
│   ├── StubGenerator.jar
│   ├── backend.native.jar
│   ├── callbacks
│   │   └── shared
│   │       └── libcallbacks.so
│   ├── clangstubs
│   │   └── shared
│   │       └── libclangstubs.so
│   ├── helpers.jar
│   ├── klib.jar
│   ├── kotlin-compiler.jar
│   ├── protobuf-java-2.6.1.jar
│   └── utilities.jar
└── nativelib
    ├── libcallbacks.so
    ├── libclangstubs.so
    ├── libllvmstubs.so
    └── liborgjetbrainskotlinbackendkonanhashstubs.so

6 directories, 16 files
```

Windows 平台上的 konan 目录文件如下：

```
kotlin-native-windows-0.3$ tree konan
konan
├── konan.properties
├── lib
│   ├── Indexer.jar
│   ├── Runtime.jar
│   ├── StubGenerator.jar
│   ├── backend.native.jar
│   ├── callbacks
│   │   └── shared
│   │       └── callbacks.dll
│   ├── clangstubs
│   │   └── shared
│   │       └── clangstubs.dll
│   ├── helpers.jar
│   ├── klib.jar
│   ├── kotlin-compiler.jar
│   ├── protobuf-java-2.6.1.jar
│   └── utilities.jar
└── nativelib
```

```
    ├── callbacks.dll
    ├── clangstubs.dll
    ├── llvmstubs.dll
    └── orgjetbrainskotlinbackendkonanhashstubs.dll

6 directories, 16 files
```

3. klib 目录

klib 目录下是 Kotlin 标准库的关联元数据文件以及 Kotlin Native 针对各个目标平台的 bc 文件：

```
kotlin-native-macos-0.3$ tree klib
klib/
└── stdlib
    ├── linkdata
    │   ├── module
    │   ├── package_konan
    │   ├── package_konan.internal
    │   ├── package_kotlin
    │   ├── package_kotlin.annotation
    │   ├── package_kotlin.collections
    │   ├── package_kotlin.comparisons
    │   ├── package_kotlin.coroutines
    │   ├── package_kotlin.coroutines.experimental
    │   ├── package_kotlin.coroutines.experimental.intrinsics
    │   ├── package_kotlin.experimental
    │   ├── package_kotlin.internal
    │   ├── package_kotlin.io
    │   ├── package_kotlin.properties
    │   ├── package_kotlin.ranges
    │   ├── package_kotlin.reflect
    │   ├── package_kotlin.sequences
    │   ├── package_kotlin.text
    │   ├── package_kotlin.text.regex
    │   ├── package_kotlin.util
    │   ├── package_kotlinx
    │   ├── package_kotlinx.cinterop
    │   └── root_package
    ├── manifest
    ├── resources
    └── targets
        ├── android_arm32
        │   ├── kotlin
        │   │   └── program.kt.bc
        │   └── native
        │       ├── launcher.bc
        │       ├── runtime.bc
        │       └── start.bc
        ├── android_arm64
        │   ├── kotlin
```

```
│   │   └── program.kt.bc
│   └── native
│       ├── launcher.bc
│       ├── runtime.bc
│       └── start.bc
├── iphone
│   ├── kotlin
│   │   └── program.kt.bc
│   └── native
│       ├── launcher.bc
│       ├── runtime.bc
│       ├── start.bc
│       ├── start.kt.bc
│       └── stdlib.kt.bc
└── macbook
    ├── kotlin
    │   └── program.kt.bc
    └── native
        ├── launcher.bc
        ├── runtime.bc
        └── start.bc

16 directories, 42 files
```

上面的目录是 kotlin-native-macos-0.3 平台的版本。可以看出，在 Mac OS 上，我们可以使用 Kotlin Native 编译 android_arm32、android_arm64、iphone、macbook 等目标平台的机器码可执行的程序。

另外，对应的 Linux 平台的目录文件如下：

```
kotlin-native-linux-0.3$ tree klib
klib/
└── stdlib
    ├── linkdata
    │   ├── module
    │   ├── package_konan
    │   ├── package_konan.internal
    │   ├── package_kotlin
    │   ├── package_kotlin.annotation
    │   ├── package_kotlin.collections
    │   ├── package_kotlin.comparisons
    │   ├── package_kotlin.coroutines
    │   ├── package_kotlin.coroutines.experimental
    │   ├── package_kotlin.coroutines.experimental.intrinsics
    │   ├── package_kotlin.experimental
    │   ├── package_kotlin.internal
    │   ├── package_kotlin.io
    │   ├── package_kotlin.properties
    │   ├── package_kotlin.ranges
    │   ├── package_kotlin.reflect
```

```
│   ├── package_kotlin.sequences
│   ├── package_kotlin.text
│   ├── package_kotlin.text.regex
│   ├── package_kotlin.util
│   ├── package_kotlinx
│   ├── package_kotlinx.cinterop
│   └── root_package
├── manifest
├── resources
└── targets
    ├── android_arm32
    │   ├── kotlin
    │   │   └── program.kt.bc
    │   └── native
    │       ├── launcher.bc
    │       ├── runtime.bc
    │       └── start.bc
    ├── android_arm64
    │   ├── kotlin
    │   │   └── program.kt.bc
    │   └── native
    │       ├── launcher.bc
    │       ├── runtime.bc
    │       └── start.bc
    ├── linux
    │   ├── kotlin
    │   │   └── program.kt.bc
    │   └── native
    │       ├── launcher.bc
    │       ├── runtime.bc
    │       └── start.bc
    └── raspberrypi
        ├── kotlin
        │   └── program.kt.bc
        └── native
            ├── launcher.bc
            ├── runtime.bc
            ├── start.bc
            ├── start.kt.bc
            └── stdlib.kt.bc

16 directories, 42 files
```

也就是说我们可以在 Linux 平台上编译 android_arm32、android_arm64、linux、raspberrypi 等平台上的目标程序。

对应 Windows 平台的目录文件如下：

```
kotlin-native-windows-0.3$ tree klib
klib/
```

```
└── stdlib
    ├── linkdata
    │   ├── module
    │   ├── package_konan
    │   ├── package_konan.internal
    │   ├── package_kotlin
    │   ├── package_kotlin.annotation
    │   ├── package_kotlin.collections
    │   ├── package_kotlin.comparisons
    │   ├── package_kotlin.coroutines
    │   ├── package_kotlin.coroutines.experimental
    │   ├── package_kotlin.coroutines.experimental.intrinsics
    │   ├── package_kotlin.experimental
    │   ├── package_kotlin.internal
    │   ├── package_kotlin.io
    │   ├── package_kotlin.properties
    │   ├── package_kotlin.ranges
    │   ├── package_kotlin.reflect
    │   ├── package_kotlin.sequences
    │   ├── package_kotlin.text
    │   ├── package_kotlin.text.regex
    │   ├── package_kotlin.util
    │   ├── package_kotlinx
    │   ├── package_kotlinx.cinterop
    │   └── root_package
    ├── manifest
    ├── mingw
    │   ├── kotlin
    │   │   └── program.kt.bc
    │   └── native
    │       ├── launcher.bc
    │       ├── runtime.bc
    │       └── start.bc
    ├── resources
    └── targets
        └── mingw
            ├── kotlin
            │   └── program.kt.bc
            └── native
                ├── launcher.bc
                ├── runtime.bc
                └── start.bc

10 directories, 32 files
```

在 Windows 平台中，Kotlin Native 是使用 mingw 库来实现的。目前，在 v0.3 预发布版本，我们在 Windows 平台上可以体验的东西比较少，像 Android、iOS、Raspberry Pi 都还不支持。



MinGW 是 Minimalist GNU for Windows 的缩写。它是一个可自由使用和自由发布的 Windows 特定头文件和使用 GNU 工具集导入库的集合，允许在 GNU/Linux 和 Windows 平台生成本地的 Windows 程序而不需要第三方 C 运行时库。MinGW 是一组包含文件和端口库，其功能是允许控制台模式的程序使用微软的标准 C 运行时库（MSVCRT.DLL），该库在所有的 NT OS 上有效，在所有的 Windows 95 发行版以上的 Windows OS 有效。使用基本运行时，你可以使用 GCC 写控制台模式的符合美国标准化组织（ANSI）的程序，可以使用微软提供的 C 运行时扩展。与基本运行时相结合，就可以既使用 CRT 又使用 Windows API 功能。

4. samples 目录

samples 目录下面是官方给出的一些实例。关于这些实例的文档介绍地址是 https://github.com/JetBrains/kotlin-native/tree/master/samples。想更加深入了解学习的读者可以参考。

16.5 本章小结

现在我们可以把 Kotlin 像 C 一样地直接编译成机器码来运行，这样在 C 语言出现的地方（例如应用于嵌入式等对性能要求比较高的场景），Kotlin 也来了。Kotlin 将会在嵌入式系统和物联网、数据分析和科学计算、游戏开发、服务端开发和微服务等领域持续发力。

Kotlin 整个语言的架构不可谓不宏大：上得了云端（服务端程序），下得了手机端（Kotlin/Native），写得了前端（JS、HTML DSL 等），嵌得了冰箱（Kotlin Native）。Kotlin 俨然已成为一门擅长多个领域的语言了。

在互联网领域，目前 Web 服务端应用开发、Android 移动端开发是 Kotlin 最活跃的领域。Kotlin 将会越来越多地进入 Java 程序员们的视野，Java 程序员也会逐渐爱上 Kotlin。

未来的可能性有很多。但是真正的未来还需要我们去创造。

本章示例代码：https://github.com/EasyKotlin/chatper16_kotlin_native_helloworld

作者简介

陈光剑

资深开发人员，目前就职于阿里巴巴集团，参与过多种平台工具的开发，具有丰富的平台开发经验，操练过Java、Android、Scala、Groovy等语言进行领域建模、架构设计、工具开发，在工作中积累了大量经验，善于总结，乐于分享，发布过大量技术文章。

封面设计：陈潇然

本书是不可多得的优秀图书，内容翔实，描述简洁，作者对编程语言有深刻的理解，融入了自己多年的开发经验……Kotlin 非常新，而且发展也非常快，是一门非常有前景的开发语言，我希望 Android 和 Java 程序员都能停下来阅读这本 Kotlin 的图书，以系统地了解这项技术。

雷卷　阿里巴巴 70 后 Java 程序员

这是一本精彩有趣的好书。阅读本书，你会再次感受到编程的乐趣。我读过许多编程的书籍，但是这本书对程序设计的深刻见解依然深深触动了我。

雷影　阿里巴巴技术专家

光剑的这本书是非常不错的 Kotlin 教程。书中有大量有趣而实用的代码实例，理论与实践相结合，强烈推荐大家阅读。

木羽　阿里巴巴高级技术专家

身处互联网大潮汹涌席卷的时代里，对新知识的无限渴望和孜孜不倦的探求，是一名技术人员对自己的基本要求。陈光剑的这本书是 Kotlin 语言领域中非常值得借鉴的学习资料，无论是刚入门的技术小白，还是需要进阶的技术痴汉，都可在这本书中找到另一番天地。

郁磊　有赞测试专家

Kotlin 具备简洁的语言、与 Java 的无缝互译、很棒的 IDE 支持，且不断进化，如果你刚刚从事或者一直从事 Android 开发，Kotlin 都将成为你很好的起点或新的助手。本书作为对 Kotlin 语言编程上手和基本使用都是不错的参考资料，必定对大家有所帮助。

石舟　阿里巴巴测试专家

在快速发展与创新的今天，不断孕育出各种新语言。Kotlin 非常具有代表性，具有简明性和独特的表达能力，同时易于“并发编程”。这本书丰富而全面地介绍了这门优秀的语言。任何对编程语言感兴趣的程序员都不妨阅读一下，相信你定会有收获。

林文国　二维火资深架构师

作为 JetBrains IDE 的忠实用户，也一样喜欢上了它的 Kotlin，而这本书绝对是 Kotlin 难得的教程，内容全面，理论和实战兼得。对我来说，读这本书的收获比自己想象的更多。

刘鼎亮　挖财资深工程师

我清楚地记得当初从 Eclipse 切换到 JetBrains 的 Intelij 时带来的效率提升和极致的用户体验，其中细节只有真正用过才能体会到。可以想见，JetBrains 出品的 Kotlin 必然也会带来各种令人惊喜的提升。光剑在许多人还没注意到 Kotlin 的时候就已经开始动手实践了，并带来了一本语言精练、内容全面、示例丰富的教程，他亲自使用并检验过 Kotlin 的很多特性，有许多心得和深刻理解，这是一本不可多得的带领程序员领略 Kotlin 的便捷和高效的指南。

有山　阿里巴巴技术专家

这是一本 Kotlin 入门的好书，内容翔实全面，并且由浅入深，通过大量的案例，引导读者一步步了解 Kotlin 的方方面面，于开卷者大有裨益。

张立柱　策道科技 CTO

上架指导：计算机/程序设计
ISBN 978-7-111-57993-9
9 787111 579939
定价：79.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn